博 士 論 文

定理証明器 HOL におけるオブジェクト指向理論の構築

指導教官　片山 卓也 教授

北陸先端科学技術大学院大学
情報科学研究科情報システム学専攻

矢竹 健朗

2006 年 3 月

要 旨

オブジェクト指向開発法の上流工程では，UML に代表されるモデリング言語により，分析モデルが構築される．分析モデルは図解ベースの表現であるため，解釈が曖昧であり，矛盾や誤りを含みやすいという問題がある．システムの欠陥を早期発見するという意味で，分析モデルに形式的な意味論を与え，検証を行うことは重要である．本研究の検証対象はオブジェクトの属性に関する不変表明である．属性は，一般に整数やリストなどのデータ型により表現されており，これを検証するためには定理証明器を用いることが妥当である．

本研究では，分析モデル検証の意味論として，定理証明器 HOL にオブジェクト指向理論を実装した．分析モデルは抽象型や問題領域に特化した型によって高度に抽象化されるという特徴がある．本研究では，このような分析モデルの特徴に対応するために，オブジェクトが任意の型の属性を持つことを可能とした．HOL の単純一階型システムにおいては，任意の型の属性を任意の数だけ格納するオブジェクトを一般的な型として表現するのは困難である．そこで本研究では，オブジェクト指向理論をアプリケーションの型情報に特化して自動生成するというアプローチをとる．アプリケーションに出現する型が予め与えられれば，一階の型システムでもオブジェクトを表現することが可能となる．

理論の健全性は definitional extension により保証する．これは，既存の健全な理論から定義の導入または健全な推論規則による導出のみを許してを新しい理論を構築する手法であり，HOL において理論を構築する際の標準的な手法である．具体的には，リストや組の理論を用いてオブジェクトを格納するためのヒープメモリ構造を定義し，その操作的意味論からオブジェクト指向の公理を導出する．

不変表明の検証法としてコラボレーションに基づく手法を提案する．この手法においては，コラボレーションをオブジェクト指向理論により与えられる基本的な演算子を用いた関数適用列として定義し，不変表明をシステムの初期状態からの遷移列に関する帰納法により証明する．状態遷移図と比較して，コラボレーションはクラスに横断する概念であり，複数のオブジェクトにまたがる不変表明の検証に有効である．

目 次

図 目 次

表 目 次

第 1章

はじめに

1.1 背景

オブジェクト指向開発法は，現在のソフトウェア開発の主流となっている．開発の上流工程では，UML (Unified Modeling Language [1]) に代表されるモデリング言語により，システムの分析モデルが構築される．分析モデルは直感的に分かりやすい図解ベースの表現を基本としているが，その代償として，解釈が曖昧になりがちであり，矛盾や誤りを含みやすいという問題がある．この誤りが開発の下流工程で発見された場合，修復には非常に大きなコストを要し，また，発見されない場合はシステムに大きな欠陥をもたらすことになる．分析モデルが矛盾や誤りを含まず，要求仕様を正確に反映していることを保証するためには，分析モデルに対し形式的な意味論を与え，検証を行う必要がある．

分析モデルに対しては，シンタックスの観点からも，意味論の観点からも，多くの検証すべき項目が取り上げられているが，本論文では，モデルの意味論に関する検証，特に，オブジェクトの属性値に関する不変表明の検証に焦点を当てる．不変表明とは，システムのいかなる状態においても成立することが期待される性質である．例えば，エアコン制御システムの場合は，温度計の値が常に 30 度以下であるといった性質，また，銀行システムの場合は，貯蓄残高が常に 0 以上であるといった性質である．このような不変表明は，整数や自然数，リストなどのデータ型に関する性質であり，検証には無限の状態空間を扱うことが可能な定理証明が有効である．

1.2 問題点

現在，実用的に使われている定理証明器は高階述語論理に基づくものであり，代表的なものには，HOL，Isabelle/HOL，PVS，Coq などがある．しかし，これらの中で，クラスや属性，継承などのオブジェクト指向概念を標準的に扱えるものはない．したがって，検証の前段階として，そのような概念を扱える理論を高階述語論理の上に導入することが必要となる．

実際，これまで，多くのオブジェクト指向理論が高階述語論理の定理証明器に実装されてきた．その多くは Java プログラム検証の意味論として実装されたものである．有力な研究には，LOOP(Isabelle/HOL, PVS)[7]，Bali(Isabelle/HOL)[9]，Krakatoa(Coq)[10] などがある．

本研究の検証対象はプログラムではなく，分析モデルである．プログラムと分析モデルの決定的な違いは「型の広がり」である．Java に出現する型は，int, bool, array といった基本的なものに限られるが，分析モデルにおいては，set, bag, stack のような抽象型や，date, time, currency などの対象領域に特化した型が出現する（図 1.1）．このような型を用いて柔軟に抽象化されるのが分析モデルである．したがって，基本的な型のみしか扱えない既存の Java プログラム検証のための理論は分析モデルの検証にはふさわしくなく，より広範な型を扱える理論が必要となる．

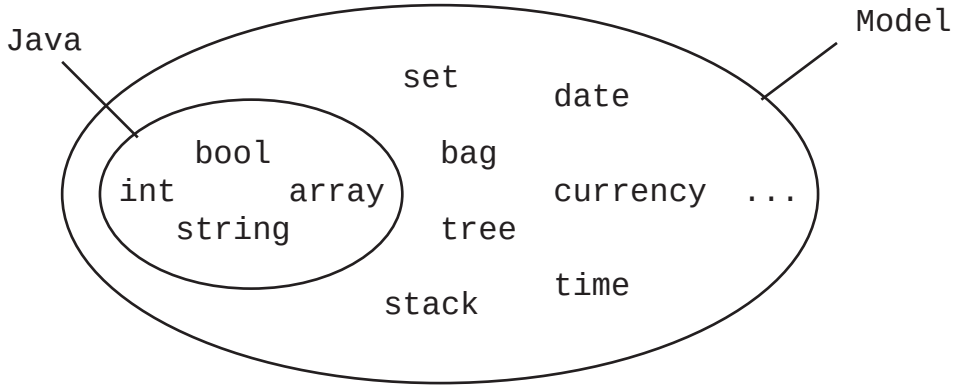


図 1.1: プログラムと分析モデルの違い

1.3 目的

本研究の目的は，分析モデルに出現しうる任意の型を属性の型とできるようなオブジェクト指向理論を定理証明器 HOL に実装することである．HOL は強力なデータ型構築機能，及び，多くのデータ型ライブラリを持ち，多様なデータ型が出現する分析モデルの検証に有効である．

実装における問題点として，任意の型の属性を持つオブジェクトをどのように HOL の単純一階型システムで表現するか，という問題がある．一般にオブジェクトは，任意の型の属性を任意個結合した型と考えることができる．このような型は，一見，複数の型変数の直積型として $(\alpha * \beta * \gamma * ...)$ のように素直に表現可能と考えられる．しかし，オブジェクトは任意個の属性を持ちうるため，いくつの要素に対して直積をとればよいかが予め分からない．レコード型や直和型でも同様である．これを解決する方法としては，extensible record という概念を用いる手法が提案されているが，オブジェクトに参照としての性質を持たせるまでには至っていない．このようなことから，オブジェクト指向の概念を扱うためには HOL よりも高度な型システムの開発が必要であるいわれており，object calculus[5] のようなオブジェクトを専門に扱うための型システムの研究が幅広く行われている．しかし，現時点では，そのような型システムを標準的に実装している定理証明器は存在しないため，既存の HOL の型システムになんらかの工夫をしてオブジェクト指向概念を実装しなければならない．

1.4 アプローチ

この問題に対処する方法として，本研究では，オブジェクト指向理論をアプリケーションの型情報が含まれるクラスモデルをもとに自動生成する，というアプローチをとる．つまり，オブジェクトを一般的な型で表現するかわりに，アプリケーションに特化した型として自動生成するというアプローチである．基本的な発想は，属性の型が入力として予め与えられれば，それを格納するオブジェクトは $(num * int * bool)$ のように直積により表現することができるということである．

本研究では，クラスモデルからそのモデルに特化したオブジェクト指向理論を

自動生成する理論生成器を実装した．理論は definitional extension により構築される．これは，conservative extension とも呼ばれ，HOL において健全な理論を構築する標準的な手法である．具体的には，既存の健全な理論から，定義の導入または健全な推論規則による導出のみを許して新しい理論を構築していく手法である．理論生成器は，HOL の既存の num, pair, list といった理論によって，オブジェクトを格納するためのヒープメモリ構造を定義し，その操作的意味論からオブジェクト指向理論を導出する．直積によって定義されるオブジェクトをヒープメモリに格納することにより，サブタイピングや参照のメカニズムをも表現することができる．結果として導出されるオブジェクト指向理論の上ではオーバーライド，動的束縛などを含めた基本的なオブジェクト指向概念を表現することができる．また，非常にシンプルで可読性の高い理論となっている．

1.5 論文の構成

本論文の構成は以下の通りである．第 2 章では，オブジェクト指向理論の定義を述べる．第 3 章では，オブジェクト指向理論の HOL における実装を述べる．第 4 章では，オブジェクト指向理論の応用例として，コラボレーションに基づく不変表明の検証法について述べる．ここでは図書館システムの不変表明の検証を行う．第 5 章では理論生成器を提供する HOL ライブラリについて述べる．第 6 では，関連研究を紹介し，本論文で提案する手法と比較する．最後に，第 7 において本論文のまとめを行う．

第 2章

オブジェクト指向理論

オブジェクト指向理論は，任意の型を許容するため，アプリケーションのクラスモデルに特化して定義される．HOL においては shallow embedding として実装される．つまり，クラスモデルの要素を，直接，HOL の型や定数によって表現し，公理を導入することにより定義される．

本章の構成は以下の通りである．まず，2.1 節では，オブジェクト指向理論の概要を例を用いて述べる．次に，2.2 節ではクラスモデルの定義を述べ，2.3, 2.4, 2.5 節ではオブジェクト指向理論の定義を述べる．2.6 節では理論におけるオブジェクト指向概念の表現法を述べる．最後に，2.7 節において考察を行う．

2.1 概要

オブジェクト指向理論は，アプリケーションのクラスモデルに依存して定義される．クラスモデルは，UML のクラス図と同様の表現であり，クラスや属性，継承関係など，アプリケーションの静的な構造を定義するモデルである．図 2.1 にクラスモデルの例を示す．`fig` は図形クラスであり，その x 座標，y 座標を表す `int` 型の属性 `x`，`y` を持つ．`rect` は長方形クラスであり，その幅，高さを表す `num` 型属性 `w`，`h` を持つ．`crect` は色つき長方形クラスであり，その色を表す `color` 型属性 `c` を持つ．`color` 型はいくつかの色を要素に持つ列挙型である．

オブジェクト指向理論はストアという概念を基本として定義される．ストアは，システムに存在する全オブジェクトのデータを保持する環境であり，型 `store` で

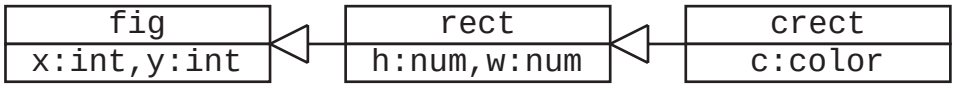


図 2.1: クラスモデルの例

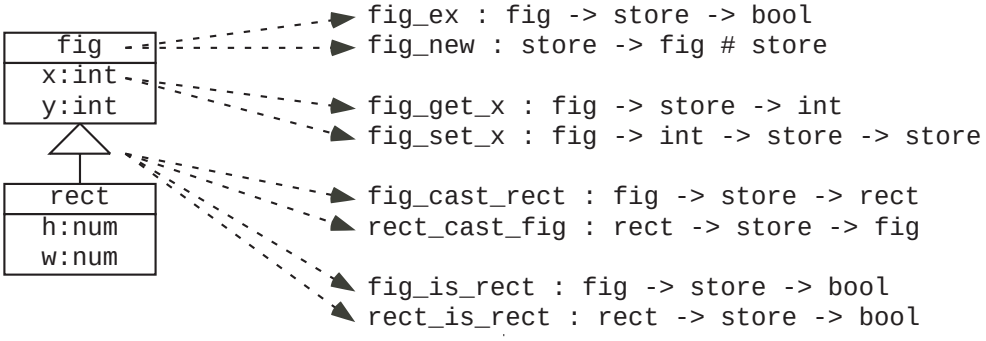
定義される．このような環境を導入するのは，オブジェクトを参照として表現するためであり，オブジェクト間でのメソッド呼び出しを実現することができる．

オブジェクトはストアに格納される自身のデータへの参照であり，その属すクラス名を名前とする型により定義される．例えば，クラス fig に属すオブジェクトの型は fig である．オブジェクトの型は「見かけ」の型であり，後で説明する型変換演算子により別の型に変換可能である．

クラスモデルの要素に対応して，数種類の演算子が理論に導入される．図 2.2 により，その対応関係を示す．クラス fig に対応して，２つの演算子 fig_new，fig_ex が導入される．それぞれ，新しい fig オブジェクトをストアに生成する演算子，fig オブジェクトがストアに存在するかどうかを検査する演算子である．fig_new は，ストアを引数とし，新しく生成した fig オブジェクトと生成後のストアを返す．fig_ex は，fig オブジェクトとストアを引数とし，そのオブジェクトがストアに存在するかどうかの真偽値を返す．公理 [A1] は，これらの演算子に関するものであり,「新しく生成されたオブジェクトは生成後のストアに存在する」を意味する．

クラス fig の属性 x に対応して，その属性に対するリード 演算子 fig_get_x，ライト演算子 fig_set_x が導入される．fig_get_x は，fig オブジェクトとストアを引数とし，そのストアに格納される属性 x の値を返す．fig_set_x は，fig オブジェクト, 整数値，ストアを引数とし，属性 x の値をその整数値に書き換えた後のストアを返す．公理 [A2] は，これらの演算子に関するものであり,「fig オブジェクトがストアに存在するならば，その属性 x を v に更新直後，同じ属性を取得したとき，得られる値は v である」を意味する．

２つのクラス fig, rect の間の継承関係に対応して，型変換演算子 fig_cast_rect, rect_cast_fig，インスタンス・オブ演算子 fig_is_rect，rect_is_rect が導入される．fig_cast_rect は，fig オブジェクトを引数とし，fig 型から rect 型に変換する．rect_cast_fig は，rect オブジェクトを引数とし，rect 型から fig 型




```
[A1]|- !s. let (f,s1) = fig_new s in fig_ex f s1
[A2]|- !f v s. fig_ex f s ==> (fig_get_x f (fig_set_x f v s) = v)
[A3]|- !r s. rect_is_rect r s ==> fig_is_rect (rect_cast_fig r s) s
[A4]|- !f s. rect_ex r s ==> (fig_cast_rect (rect_cast_fig r s) = r)
```

図 2.2: クラスモデルと理論の対応

に変換する．fig_is_rectは，figオブジェクトを引数とし，それがrectクラスのインスタンスであるかどうかを検査する．rect_is_rectは，rectオブジェクトを引数とし，それがrectクラスのインスタンスであるかどうかを検査する．

オブジェクトは生成後，型変換により見かけの型が変化するが，インスタンス・オブ演算子はその生成時の型を記憶する役割を持つ．例えば，rect_newで生成されたrectクラスのインスタンスはrect_cast_figにより，見かけの型がfig型に変化するが，そのfig型のオブジェクトについてfig_is_rectが成立するため，rectクラスのインスタンスであること分かる．公理 [A3] はこれを示すものである．

公理 [A4] は型変換の可逆性に関するものであり,「rectオブジェクトがストアに存在するならば，それをrect型からfig型に変換し，もう一度，fig型からrect型に変換し直すと，もとのオブジェクトと同一となる」を意味する．

2.2 クラスモデル

クラスモデルは，システムの静的構造を定義する．具体的には，クラス名，属性名とその型，デフォルト値，及び，継承関係を定義する．

クラスモデルを以下の 6 つ組として定義する．

$$CM = (C, A, \mathcal{M}_{attr}, \mathcal{M}_{inher}, \mathcal{T}, \mathcal{V})$$

- $\mathcal{M}_{attr} : C \to Pow(A)$
- $\mathcal{M}_{inher} : C \to Pow(C)$
- $\mathcal{T} : C \times A \to Type$
- $\mathcal{V} : C \times A \to Value$

C, Aはそれぞれ, システムに出現するクラス名の集合, 属性名の集合である. $\mathcal{M}_{attr}$は，クラスとその属性集合を対応付ける写像である．$\mathcal{M}_{inher}$は，クラスと，その子クラス集合を対応付ける写像である．継承関係は，Java のような単一継承であり，木構造をなす．ただし，ルートクラスは 1 つとは限らず，複数の継承木が存在してもよい．$\mathcal{T}$は，クラスと属性に対し，その属性の型を対応付ける写像である．集合$Type$は，HOL における型変数を含まない任意の型の集合である．$c \in C$と同名の型が$Type$に存在するとし，クラス名によりそのクラスに属すオブジェクトの型を表す．$\mathcal{V}$は，クラスと属性に対し，その属性のデフォルト値を対応付ける写像である．集合$Value$は$Type$に含まれるすべての型の要素の集合である．

以下に，いくつかの記号を導入する．

$c \lhd d$により，cがdの親クラスであることを表す（UML の継承の三角記号から連想）．つまり，$c \lhd d = d \in \mathcal{M}_{inher}(c)$である．さらに，$c \lhd^+ d$は，$c$が$d$の祖先クラスであることを，$c \lhd^* d$は，$c = d$または$c \lhd^+ d$であることを意味する．例えば，$\texttt{fig} \lhd \texttt{rect}$，$\texttt{fig} \lhd^+ \texttt{crect}$，$\texttt{fig} \lhd^* \texttt{fig}$である．

$attr(c)$により，クラスcの属性と，継承された属性の集合を表す．つまり，$attr(c) = \{a \mid a \in \mathcal{M}_{attr}(d), d \lhd^* c\}$である．例えば，$attr(rect) = \{x, y, w, h\}, attr(crect) = \{x, y, w, h, c\}$である．

$relative(c, d)$ により，クラス c, d が同一の継承木に属すことを表す．つまり，$relative(c, d) = \exists r.\ r \lhd^* c \wedge r \lhd^* d$である．

2.3 型，定数

ストアは，型$store$として定義する．$store$は定数としてEmpを持つ．これはどのクラスからもオブジェクトが生成されていない，空のストアを表す．

クラス c に属すオブジェクトは，型 c を持つとする．各 c は，定数として $Null^c$ を持つ．これは，そのクラスの NULL オブジェクトを表す．NULL オブジェクトはストアに存在しない．

クラス c の属性 a の未定義の値を意味する定数として，$Unknown_a^c : \mathcal{T}(c, a)$ を導入する．これはストアに存在しないオブジェクトに対し次節で定義される get 演算子による属性取得を行った場合に返される値である．

HOL において，$store$, Emp, $Null^c$, $Unknown_a^c$ はそれぞれ，`store`, `store_emp`, `c_null`, `c_unknown_a` と表記する．

2.4 演算子

6 種類の基本演算子を以下に定義する．

- **new 演算子**

  $New^c : store \to c * store \quad (c \in C)$

  クラス c のオブジェクトを生成する関数．$New^c\ s = (o, s')$ であるとき，o は新しく生成された c クラスのオブジェクト，s' は生成後のストアである．

- **ex 演算子**

  $Ex^c : c \to store \to bool \quad (c \in C)$

  クラス c のオブジェクトがストアに存在するかどうかを検査する述語．$Ex^c\ o\ s = x$ であるとき，x はオブジェクト o がストア s に存在するかどうかの真偽値である．

- **get 演算子**

  $Get_a^c : c \to store \to \mathcal{T}(c, a) \quad (c \in C, a \in attr(c))$

  クラス c のオブジェクトの属性 a を取得する関数．$Get_a^c\ o\ s = x$ のとき，x はオブジェクト o の属性 a のストア s における値である．

- **set 演算子**

$Set_a^c : c \to \mathcal{T}(c, a) \to store \to store \quad (c \in C, a \in attr(c))$

クラス c のオブジェクトの属性 a を更新する関数．$Set_a^c\ o\ x\ s = s'$ のとき，s' はストア s においてオブジェクト o の属性 a を x に更新して得られるストアである．

- **cast 演算子**

$Cast_d^c : c \to store \to d \quad (c, d \in C, c \lhd^+ d \text{ or } d \lhd^+ c)$

クラス c のオブジェクトをクラス d のオブジェクトに型変換する関数．$Cast_d^c\ o\ s = o'$ のとき，o' は c 型のオブジェクト o をストア s において型変換して得られる d 型のオブジェクトである．

- **is 演算子**

$Is_d^c : c \to store \to bool \quad (c, d \in C, c \lhd^* d)$

クラス c のオブジェクトがクラス d のインスタンスであるかを検査する述語．$Is_d^c\ o\ s = x$ のとき，x はストア s においてオブジェクト o がクラス d のインスタンスであるかどうかの真偽値である．

HOL において，Ex^c, New^c, Get_a^c, Set_a^c, $Cast_d^c$, Is_d^c はそれぞれ，`c_ex`, `c_new`, `c_get_a`, `c_set_a`, `c_cast_d`, `c_is_d` と表記する．

2.5 公理

以上で定義した演算子に関する公理を以下に導入する．また，公理と定数，演算子の関係を表 2.1, 2.2 に示す．

1. **NOT_EX_EMP**

$$\forall o.\ \neg(Ex^c\ o\ Emp)$$

ストアの初期値 Emp にはオブジェクトは存在しない.

2. **NOT_EX_NULL**

$$\forall s.\ \neg(Ex^c\ Null^c\ s)$$

NULL オブジェクトはストアに存在しない.

3. **EX_IS**

$$\forall o\ s.\ Ex^c\ o\ s = Is^c_{d_1}\ o\ s \vee ... \vee Is^c_{d_n}\ o\ s\ (\{d_1, ..., d_n\} = \{d \mid c \lhd^* d\})$$

ストアに存在するクラス c のオブジェクト o は，c または，その子孫クラスのいずれかのインスタンスである.

4. **NOT_EX_FST_NEW**

$$\forall s.\ \neg(Ex^c\ (Fst\ (New^c\ s))\ s)$$

新しく生成されたオブジェクトは生成前のストア s に存在しない.

5. **NOT_EX_FST_NEW_CAST**

$$\forall s.\ \neg(Ex^c\ (Cast^c_d\ (Fst\ (New^c\ s))\ (Snd\ (New^x\ s)))\ s)$$

新しく生成されたオブジェクトを型変換して得られるオブジェクトは，生成前のストア s に存在しない.

6. **IS_IMP_NOT_IS**

$$\forall o\ s.\ Is^c_d\ o\ s \Rightarrow \neg(Is^c_e\ o\ s)\ \ (d \neq e)$$

クラス c のオブジェクト o がクラス d のインスタンスであれば，d とは異なるクラス e のインスタンスではない.

7. **IS_CAST**

$$\forall o\ s.\ Is_e^c\ o\ s \Rightarrow Is_e^d\ (Cast_d^c\ o\ s)\ s \quad (d \lhd^+ e)$$

クラス c のオブジェクト o がクラス e のインスタンスであれば，o を e よりも祖先であるいずれのクラス d に型変換したとしても，そのオブジェクトは e のインスタンスである．つまり，生成時の型は，型変換により見かけの型が変化しても，一定である．

8. **IS_NEW**

$$\forall o\ s.\ Is_c^c\ o\ (Snd\ (New^c\ s)) = (o = Fst\ (New^c\ s)) \vee Is_c^c\ o\ s$$

クラス c のオブジェクトを生成後，オブジェクト o がクラス c のインスタンスであることは，o がその新しく生成されたオブジェクトであるか，または，生成前のストアにおいてすでに c のインスタンスであったオブジェクトであることと同値である．

9. **IS_NEW_CAST**

$$\forall o\ s.\ Is_d^c\ o\ (\#2(New^d\ s)) =$$
$$(o = Cast_c^d\ (Fst\ (New^d\ s))\ (Snd\ (New^d\ s)) \vee Is_d^c\ o\ s$$

クラス d のオブジェクトを生成後，d の祖先クラス c のオブジェクト o が d のインスタンスであることは，o がその新しく生成されたオブジェクトを c に型変換して得られたオブジェクトであるか，または，生成前のストアにおいてすでに d のインスタンスであった c オブジェクトであることと同値である．

10. **DIFF_IS_NEW**

$$\forall o\ s.\ Is_d^c\ o\ (Snd\ (New^e\ s)) = Is_d^c\ o\ s \quad (d \neq e)$$

クラス c のオブジェクトがクラス d のインスタンスであることは，d とは異なるクラス e のオブジェクトの生成には無関係である．

11. **IS_SET**

$$\forall o_1\ o_2\ x\ s.\ Is_d^c\ o_1\ (Set_a^e\ o_2\ x\ s) = Is_d^c\ o_1\ s$$

あるオブジェクトがどのクラスのインスタンスであるかは，属性の更新には無関係である．

12. **DOWN_NULL**

$$\forall o\ s.\ Is_c^c\ o\ s \Rightarrow (Cast_d^c\ o\ s = Null^d)\ \ (c \lhd^+ d)$$

クラス c のインスタンスをそれよりも子孫のクラス d に型変換すれば，変換先のクラスの NULL オブジェクトとなる．

13. **NOT_EX_CAST**

$$\forall o\ s.\ \neg(Ex^c\ o\ s) \Rightarrow (Cast_d^c\ o\ s = Null^d)$$

ストアに存在しないオブジェクトを型変換すれば変換先のクラスの NULL オブジェクトとなる．

14. **UP_11**

$$\forall o_1\ o_2\ s.\ Ex^d\ o_1\ s \wedge Ex^d\ o_2\ s \Rightarrow$$
$$\neg(o_1 = o_2) \Rightarrow \neg(Cast_c^d\ o_1\ s = Cast_c^d\ o_2\ s)\ \ (c \lhd^+ d)$$

ストアに存在するクラス d の 2 つのオブジェクト o_1，o_2 について，それらが異なるオブジェクトならば，それらを祖先クラス c に型変換しても異なるオブジェクトとなる．

15. **DOWN_11**

$$\forall o_1\ o_2\ s.\ Is_e^c\ o_1\ s \wedge Is_e^c\ o_2\ s \Rightarrow$$
$$\neg(o_1 = o_2) \Rightarrow \neg(Cast_d^c\ o_1\ s = Cast_d^c\ o_2\ s)\ \ (c \lhd^+ d \text{ and } d \lhd^* e)$$

クラス e のインスタンスであるクラス c の 2 つのオブジェクト o_1，o_2 について，それらが異なるオブジェクトならば，それらを c の子孫であり e の祖先であるクラス d に型変換しても異なるオブジェクトとなる．

16. **UP_DOWN**

$$\forall o\ s.\ Ex^d\ o\ s \Rightarrow (Cast^c_d\ (Cast^d_c\ o\ s)\ s = o)\ \ (c \lhd^+ d)$$

クラス d のオブジェクト o がストアに存在するならば，それを祖先クラス c に型変換し，もう一度，d に型変換すれば，もとのオブジェクト o 自身となる．

17. **DOWN_UP**

$$\forall o\ s.\ Ex^d\ (Cast^c_d\ o\ s)\ s \Rightarrow (Cast^d_c\ (Cast^c_d\ o\ s)\ s = o)\ \ (c \lhd^+ d)$$

クラス c のオブジェクト o を子孫クラス d に型変換可能（型変換して得られるオブジェクトがストアに存在する）ならば，それを子孫クラス d に型変換し，もう一度，c に型変換すれば，もとのオブジェクト o 自身となる．なお，o を c から d に型変換可能であるのは，o が d または，d よりも子孫のクラスのインスタンスである．

18. **CAST_CAST**

$$\forall o\ s.\ Cast^d_e\ (Cast^c_d\ o\ s)\ s = Cast^c_e\ o\ s$$
$$((c \lhd^+ d \text{ and } d \lhd^+ e) \text{ or } (e \lhd^+ d \text{ and } d \lhd^+ c))$$

これは cast 演算子の推移的な適用に関する公理である．クラス c からクラス e に型変換することは，c から一旦，継承関係において c と e の間にあるクラス d に型変換し，それを d に型変換することに等しい．

19. **CAST_SET**

$$\forall o_1\ o_2.\ Cast^c_d\ o_1\ (Set^e_a\ o_2\ x\ s) = Cast^c_d\ o_1\ s$$

型変換は属性の更新に無関係である．

20. **EX_CAST_NEW**

$$\forall o\ s.\ Ex^c\ o\ s \Rightarrow\ (Cast^c_d\ o\ (Snd\ (New^e\ s)) = Cast^c_d\ o\ s)$$
$$(c \lhd^* e \text{ and } d \lhd^* e)$$

クラス c, d がともにクラス e の子孫であるとき，c のオブジェクト o を，d へ型変換して得られるオブジェクト参照の値は，o がストアに存在するならば，e インスタンスの生成に無関係である．つまり，型変換の結果得られるオブジェクトは，すでに存在するオブジェクトであるため，新しく生成される e インスタンスを型変換して得られるオブジェクトとは異なるオブジェクトである．

21. **DIFF_CAST_NEW**

$$\forall o\ s.\ Cast^c_d\ o\ (Snd\ (New^e\ s)) = Cast^c_d\ o\ s\ \ (c \not\lhd^* e \text{ or } d \not\lhd^* e)$$

あるオブジェクト o を，クラス e のインスタンスの型変換の有効範囲（e の子孫クラス間での変換）を越えるクラス間で型変換を行った場合，得られるオブジェクト参照の値は，e インスタンスの生成に無関係である．つまり，型変換の結果得られるオブジェクトは，e インスタンスを型変換して得られるオブジェクトとは異なるオブジェクトである．

22. **NOT_EX_GET**

$$\forall o\ s.\ \neg(Ex^c\ o\ s) \Rightarrow (Get^c_a\ o\ s = Unknown^d_a)\ \ (a \in \mathcal{M}_{attr}(c))$$

ストアに存在しないオブジェクトの属性を取得した場合，その値は，その属性の未定義の値を意味する定数となる．

23. **NOT_EX_SET**

$$\forall o\ x\ s.\ \neg(Ex^c\ o\ s) \Rightarrow (Set^c_a\ o\ x\ s = s)$$

ストアに存在しないオブジェクトの属性更新は無効となる．

24. **SPR_GET**

$$\forall o\ s.\ Get^d_a\ o\ s = Get^c_a\ (Cast^d_c\ o\ s)\ s\ \ (c \lhd^+ d \text{ and } a \in attr(c))$$

クラス c が属性 a を持つとき，c の子孫クラス d のオブジェクト o の属性 a を取得して得られる値は，o を c に型変換し，そのオブジェクトについて得られる a の値と等しい．

25. **SPR_SET**

$$\forall o\ s.\ Set^d_a\ o\ x\ s = Set^c_a\ (Cast^d_c\ o\ s)\ x\ s\ \ (c \lhd^+ d \text{ and } a \in attr(c))$$

クラス c が属性 a を持つとき，c の子孫クラス d のオブジェクト o の属性 a を更新することは，o を c に型変換し，そのオブジェクトについて a を更新することに等しい．

26. **GET_SET**

$$\forall o\ s.\ Ex^c\ o\ s \Rightarrow (Get^c_a\ o\ (Set^c_a\ o\ x\ s) = x)$$

オブジェクト o がストアに存在するならば，その属性 a を x に更新し，その直後に同じ属性を取得したとき，その値は x である．

27. **DIFF_OBJ_GET_SET**

$$\forall o_1\ o_2\ s.\ \neg(o_1 = o_2) \Rightarrow (Get^c_a\ o_1\ (Set^c_a\ o_2\ x\ s) = Get^c_a\ o_1\ s)$$

オブジェクト o_2 の属性 a を更新直後，それとは異なるオブジェクト o_1 の同じ属性を取得したとき，その値は，更新前に得られる値と等しい．つまり，異なる 2 つのオブジェクトの属性は独立している．

28. **DIFF_GET_SET**

$$\forall o_1\ o_2\ s.\ Get^c_a\ o_1\ (Set^d_b\ o_2\ x\ s) = Get^c_a\ o_1\ s$$
$$((c \not\lhd^* d \text{ and } d \not\lhd^* c) \text{ or } a \neq b)$$

クラス c, d が継承関係にないとき，または，属性 a, b が異なるとき，d のオブジェクト o_2 の属性 b を更新直後，c のオブジェクト o_1 の属性 a を取得した

とき，その値は，更新前に得られる値と等しい．これも??と同様であり，異なる２つの属性は独立していることを意味する．

29. **GET_NEW**

$$\forall s.\ Get_a^c\ (Fst\ (New^c\ s))\ (Snd\ (New^c\ s)) = \mathcal{V}(d,a)\ \ (a \in \mathcal{M}_{attr}(d))$$

オブジェクトを生成直後，その新しいオブジェクトの属性を取得した場合，その値はその属性のデフォルト値となる．

30. **GET_NEW_CAST**

$$\forall o\ s.\ (o = Cast_c^d\ (Fst\ (New^d\ s))\ (Snd\ (New^d\ s))) \Rightarrow$$
$$(Get_a^c\ o\ (Snd\ (New^d\ s)) = \mathcal{V}(e,a))\ \ (c \lhd^+ d \text{ and } a \in \mathcal{M}_{attr}(e))$$

オブジェクトを生成直後，その新しいオブジェクトを祖先クラスに型変換し，属性を取得した場合，その値はその属性のデフォルト値となる．

31. **EX_GET_NEW**

$$\forall o\ s.\ Ex^c\ o\ s \Rightarrow (Get_a^c\ o\ (Snd\ (New^d\ s)) = Get_a^c\ o\ s)\ \ (c \lhd^* d)$$

オブジェクト生成直後，ストアにすでに存在するオブジェクトの属性を取得する場合，その値は生成前に取得する値と同一となる．つまり，oが存在すれば，oは新しく生成されるオブジェクトとは異なるオブジェクトであるから，その属性更新はオブジェクト生成とは無関係に行うことができる．

32. **DIFF_GET_NEW**

$$\forall o\ s.\ Get_a^c\ o\ (Snd\ (New^d\ s)) = Get_a^c\ o\ s\ \ (c \not\lhd^* d)$$

クラスdのオブジェクト生成直後，dの祖先ではないクラスcのオブジェクトoの属性を取得する場合，その値は生成前に取得する値と同一となる．つまり，cはdの祖先ではないため，oは新しく生成されるオブジェクトとは異なるオブジェクトであり（dインスタンスは型変換によってcオブジェクトにはなりえない），属性取得はオブジェクト生成の影響を受けない．

33. **SET_SET**

$$\forall o\ x\ y\ s.\ Set_a^c\ o\ x\ (Set_a^c\ o\ y\ s) = Set_a^c\ o\ x\ s$$

あるオブジェクトについて，属性 a を更新直後，同じ属性を更新する場合，前の更新は無効となる．

34. **DIFF_OBJ_SET_SET**

$$\forall o_1\ o_2\ x\ y\ s.\ \neg(o_1 = o_2) \Rightarrow$$
$$(Set_a^c\ o_1\ x\ (Set_a^c\ o_2\ y\ s) = Set_a^c\ o_2\ y\ (Set_a^c\ o_1\ x\ s))$$

オブジェクト o_1 の属性 a の更新と, それとは異なるオブジェクト o_2 の属性 a の更新について，その更新順序は入れ替え可能である．

35. **DIFF_SET_SET**

$$\forall o_1\ o_2\ x\ y\ s.\ Set_a^c\ o_1\ x\ (Set_b^d\ o_2\ y\ s) = Set_b^d\ o_2\ y\ (Set_a^c\ o_1\ x\ s)$$
$$((c \not\lhd^* d \text{ and } d \not\lhd^* c) \text{ or } (a \neq b))$$

属性 a の更新と，a とは異なる属性 b の更新について，その更新順序は入れ替え可能である．

36. **EX_SET_NEW**

$$\forall o\ x\ s.\ Ex^c\ o\ s \Rightarrow$$
$$(Set_a^c\ o\ x\ (Snd\ (New^d\ s)) = Snd\ (New^d\ (Set_a^c\ o\ x\ s)))\ \ (c \lhd^* d)$$

クラス c のオブジェクト o の属性の更新と，c の子孫クラス d のオブジェクトの生成は，o がストアに存在すれば，その順序は入れ替え可能である．つまり，o が存在すれば，o は新しく生成されるオブジェクトとは異なるオブジェクトであるから，その属性更新はオブジェクト生成とは無関係に行うことができる．

37. **DIFF_SET_NEW**

$$\forall o\ x\ s.\ Set_a^c\ o\ x\ (Snd\ (New^d\ s)) = Snd\ (New^d\ (Set_a^c\ o\ x\ s))\ \ (c \not\lhd^* d)$$

クラス c のオブジェクト o の属性の更新と，c の子孫ではないクラス d のオブジェクトの生成について，その順序は入れ替え可能である．つまり，c は d の祖先ではないため，o は新しく生成されるオブジェクトとは異なるオブジェクトであり（d インスタンスは型変換によって c オブジェクトにはなりえない），その属性更新はオブジェクト生成とは無関係に行うことができる．

38. **DIFF_NEW_NEW**

$$\forall s.\ Snd\ (New^c\ (Snd\ (New^d\ s))) = Snd\ (New^d\ (Snd\ (New^d\ s)))$$
$$(\text{not } relative(c, d))$$

異なる継承木に属すクラス c, d のオブジェクトの生成順序は入れ替え可能である．

39. **SET_GET**

$$\forall o\ s.\ Set_a^c\ o\ (Get_a^c\ o\ s)\ s = s$$

ある属性をそのストアにおける値で更新すれば，もとのストアに一致する．

40. **FST_NEW_SET**

$$\forall o\ x\ s.\ Fst\ (New^c\ (Set_a^d\ o\ x\ s)) = Fst\ (New^c\ s)$$

オブジェクト生成によって得られる新しいオブジェクト参照の値は属性更新の影響を受けない．

41. **DIFF_FST_NEW_NEW**

$$\forall s.\ Fst\ (New^c\ (Snd\ (New^d\ s))) = Fst\ (New^c\ s)\ \ (c \not\lhd^* d)$$

クラス c のオブジェクト生成によって得られる新しいオブジェクト参照の値は，c の子孫ではないクラス d のオブジェクト生成の影響を受けない．

2.6 オブジェクト指向概念の表現

様々なオブジェクト指向概念が，これまでに定義した演算子を用いて表現可能である．以下にその典型的な表現方法を示す．

2.6.1 メソッド

メソッドは，get 演算子，set 演算子，new 演算子，cast 演算子，及び，HOL ライブラリが提供する関数群を用いて定義される．クラス fig がメソッド move を持つとする．move は，x 座標，y 座標をそれぞれ，dx，dy だけ移動させる．このメソッドは，属性 x，y に対応して与えられる get 演算子，set 演算子と，HOL の整数ライブラリに提供される '+' を用いて，次のように定義される．

```
fig_move : fig -> int -> int -> store -> store
fig_move f dx dy s =
   let x = fig_get_x f s in
   let y = fig_get_y f s in
   let s1 = fig_set_x f (x+dx) s in
      fig_set_y f (y+dy) s1
```

2.6.2 メソッドの継承

メソッドの継承は，親クラスのメソッド定義を子クラスで複製するメカニズムである．本理論では，オブジェクトを親クラスの型に変換し，そのクラスで定義されるメソッドを呼び出すことにより実現される．クラス rect が親クラス fig のメソッド move を継承するとき，このメソッドは次のように定義される．

```
rect_move : rect -> int -> int -> store -> store
rect_move r dx dy s =
   let f = rect_cast_fig r s in
      fig_move f dx dy s
```

2.6.3 オーバーライド

オーバーライドは，親クラスのメソッドを子クラスで再定義するメカニズムである．クラス crect が親クラス rect のメソッド move をオーバーライドするとする．この move は，座標位置を移動した後，図形の色を red に変更する．これは次のように定義される．

```
crect_move : crect -> int -> int -> store -> store
crect_move c dx dy s =
   let r = crect_cast_rect c s in
   let s1 = rect_move r s dx dy s in
      crect_set_color c red s1
```

再定義中に親クラスのメソッドを呼び出すときは，メソッドの継承と同様，親クラスに変換してメソッドを呼び出す．

2.6.4 仮想関数と動的束縛

仮想関数は，いわゆる generic function であり，適用されるオブジェクトがどのクラスのインスタンスであるかに応じて，その関数本体を切り替える関数である．このメカニズムは，is 演算子と if 文により表現可能である．

v_fig_move を fig オブジェクトに適用されるメソッド move の仮想関数版であるとする．この関数は，適用される fig オブジェクトが fig, rect, crect のどのクラスのインスタンスであるかを fig_is_fig, fig_is_rect, fig_is_crect により判定し，適切なメソッドを呼び出す（図 2.3）．

```
v_fig_move : fig -> int -> int -> store -> store
v_fig_move f dx dy s =
  if fig_is_fig f s then
    fig_move f dx dy s
  else if fig_is_rect f s then
    rect_move (fig_cast_rect f s) dx dy s
  else if fig_is_crect f s then
    crect_move (fig_cast_crect f s) dx dy s
  else s
```

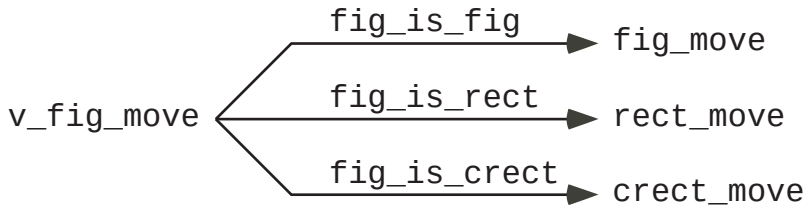


図 2.3: is 演算子による動的束縛

2.6.5 その他のオブジェクト指向概念

基本的なオブジェクト指向概念は以上のように表現することができる．それ以外にも，アクセス制限，オーバーロード，抽象メソッド，インターフェースなどがある．

public や private などのアクセス制限に関しては，直接 HOL において表現するのは困難であり，無理に実装すると理論の複雑化を招くため，ターゲット言語をオブジェクト指向理論とするような何らかのオブジェクト指向言語を定義し，その言語に対するコンパイラが処理すべきである．

オーバーロードは同じ名前のメソッドを複数回定義することであるが，これは，同じ名前の関数に複数の定義を与えるというオーバーロードの機能が HOL の関数 `overload_on` として実装されており，これを利用して実現することができる．

抽象メソッドに関しては，例えば，`fig` クラスが `move` を抽象メソッドとして宣

言している場合，以下のように，仮想関数の定義において，fig クラスの実装への分岐をなくすことにより表現できる．

```
v_fig_move f dx dy s =
  if fig_is_rect f s then
    rect_move (fig_cast_rect f s) dx dy s
  else if fig_is_crect f s then
    crect_move (fig_cast_crect f s) dx dy s
  else s
```

インターフェースの実現に関しては検討中である．

2.7 考察

2.7.1 Deep embedding vs shallow embedding

形式言語を定理証明器に実装する手法には，deep embedding と shallow embedding の 2 通りの手法が存在する [6]．前者は，言語の抽象構文を型として定義し，その型の要素を用いて意味論を定義する手法である．後者は，言語の要素を，直接，理論の要素によって表現することにより意味論を定義する手法である．前者は言語自体の性質（例えば，言語の型システムが安全である，など）の証明に適しており，後者は言語のインスタンスの性質（例えば，変数 x の値が 10 以下である，など）の証明に適している．本研究の目的は，個々のアプリケーションごとに属性の不変表明を検証することであるため，後者の実装法に従っている．つまり，クラスモデルの要素を直接 HOL の型や関数によって表現することにより，クラスモデルの意味論を定義している．

2.7.2 アプリケーションに特化することの問題点

本研究で提案するオブジェクト指向理論は，任意のデータ型を扱えるようにするため，アプリケーションの型情報に特化して自動生成される．アプリケーションに特化することの短所としては，証明される定理もアプリケーションに特化し

てしまうことである．本来，定理は一度証明すればもう二度と証明する必要がなく，何度も再利用できるものである．しかし，本理論において証明された定理は，そのアプリケーションでしか使うことができない．例えば次の定理は,「rect オブジェクト r が存在するならば，その属性 x を v に更新した直後に，r を fig 型に変換して得られるオブジェクト f について取得される属性 x の値は v となる」という定理であるが（公理 25, 26 などから導出可能），

```
|- !r v s. rect_ex r s ==>
    (fig_get_x (rect_cast_fig r s) (rect_set_x r v s) = v)
```

次のように，別のアプリケーションにおいてもこれと同様の意味を持つ定理が証明できる．

```
|- !b v s. book_ex b s ==>
    (item_get_iid (book_cast_item r s) (book_set_iid b v s) = v)
```

この定理は，前の定理の fig, rect, x をそれぞれ item, book, iid に変えただけであり，属性のアクセスに関する同様の性質を述べた定理である．このように同様の意味の定理を毎回アプリケーションごとに証明しなければならないという問題がある．

これを解決する方法は，同様の意味の定理を自動的に証明するタクティックを実装することである．タクティックは ML によりカスタマイズ可能であり，アプリケーションに依存する部分を吸収するようにプログラムすることができる．上の二つの定理は項が異なるだけであり，証明過程においても，書き換えに用いられる公理に含まれる項の部分だけを変更すれば全く同一のタクティックとして証明される．このような場合，タクティックを項を入力パラメータとし，その項に応じて書き換えに必要な公理を選択，適用するように実装すればよい．これにより，一つのタクティックで同様の定理を自動的に証明できるようになる．以下はこのようなタクティックをもとに作られた関数であり，定理を識別するためのキーとなる項を入力すると，それに対応する定理を自動的に証明し，出力する．

```
SPR_GET_SUB_SET : {Get:term, Set:term} -> thm
```

```
> SPR_GET_SUB_SET{Get:‘‘fig_get_x‘‘, Set=‘‘rect_set_x‘‘};
- val it = |- !r v s. rect_ex r s ==>
    (fig_get_x (rect_cast_fig r s) (rect_set_x r v s) = v) : thm

> SPR_GET_SUB_SET{Get:‘‘item_get_iid‘‘, Set=‘‘book_set_iid‘‘};
- val it = |- !b v s. book_ex b s ==>
    (item_get_iid (book_cast_item r s) (book_set_iid b v s) = v) : thm
```

2.7.3 例外的な場合の扱い

Javaでは，NULL参照の属性にアクセスしようとしたり，ダウンキャストにより不適切なクラスに変換しようとした場合，それぞれ，例外NullPointerException, ClassCastExceptionが発生する．オブジェクト指向理論においてもこのような不正な場合が起こりうるが，HOLは全域関数の体系であるため，例外的な入力に対してもその出力の値が定義されていなければならない．本研究では例外的な場合を以下のように解決している．

- NULLオブジェクト，及び，ストアに存在しないオブジェクトに対する属性のアクセスは，属性取得の場合，定数$Unknown_a^c$を出力，属性更新の場合，ストアを更新しないでそのまま出力する（公理22, 23）．
- 不正な型変換に対しては，変換先のクラスのNULLオブジェクトを返す（公理12）．

このように定義したのは理論のシンプルさ，可読性を重視したためである．問題があるとすれば，正常な実行の流れにおいて不正な実行を検出できないことである．

これはオプション型$'a\ option$を用いて解決することができる．つまり，不正な入力に対しては$NONE$を返し，正常な入力に対しては$SOME\ x$を返す．このようにすれば，正常な出力を不正な出力を区別することができる．ただし，正常な属性値やストアに毎度$SOME$を付加しなければならず，可読性は低下する．現時点では，理論の可読性を優先する選択をしている．例外の概念が必要になれば，オプション型を用いて現在のオブジェクト指向理論に上乗せすればよい．

Axioms	New	Ex	Get	Set	$Cast$	Is	$Null$	Emp
NOT_EX_EMP		○						○
NOT_EX_NULL		○					○	
EX_IS		○				○		
NOT_EX_FST_NEW	○	○						
NOT_EX_FST_NEW_CAST	○	○			○			
IS_IMP_NOT_IS						○		
IS_CAST					○	○		
IS_NEW	○					○		
IS_NEW_CAST	○				○	○		
DIFF_IS_NEW	○					○		
IS_SET				○		○		
DOWN_NULL					○	○	○	
NOT_EX_CAST		○			○		○	
UP_11		○			○			
DOWN_11					○	○		
UP_DOWN		○			○			
DOWN_UP		○			○			
CAST_CAST					○			
EX_CAST_NEW	○	○			○			
DIFF_CAST_NEW	○				○			

表 2.1: 公理対応表 (1)

Axioms	New	Ex	Get	Set	$Cast$	Is	$Null$	$Unknown$
NOT_EX_GET		○	○					○
NOT_EX_SET		○		○				
SPR_GET			○		○			
SPR_SET				○	○			
GET_SET		○	○	○				
DIFF_OBJ_GET_SET			○	○				
DIFF_GET_SET			○	○				
GET_NEW	○		○					
GET_NEW_CAST	○		○		○			
EX_GET_NEW	○	○	○					
DIFF_GET_NEW	○		○					
SET_SET				○				
DIFF_OBJ_SET_SET				○				
DIFF_SET_SET				○				
EX_SET_NEW	○	○		○				
DIFF_SET_NEW	○			○				
DIFF_NEW_NEW	○							
SET_GET			○	○				
FST_NEW_SET	○			○				
DIFF_FST_NEW_NEW	○							

表 2.2: 公理対応表 (2)

第 3章

オブジェクト指向理論の実装

本研究では，クラスモデルからそのモデルに特化したオブジェクト指向理論を自動生成する理論生成器を実装した．理論は definitional extension により構築され，健全性が保証される．オブジェクト指向理論の構築は，まず，オブジェクトを格納するためのヒープメモリ構造を自然数やリストといった基礎的な理論によって定義し，次に，その操作的意味論から公理を導出するという方法で行った．

本章の構成は以下の通りである．まず，3.1 節において実装の概要を例を用いて述べる．次に，3.2 節においてヒープメモリの一般的な実装法を述べる．最後に，3.3 節においてヒープメモリからのオブジェクト指向理論を導出法を述べる．

3.1 概要

まず，ストアの構造について説明する．ストアはオブジェクトのデータを格納するためのヒープメモリとして実装される．図 3.1 は，図 2.1 の例におけるヒープメモリのスナップショットである．ヒープメモリ全体は，3 つのクラス `fig`，`rect`，`crect` に対応する 3 つのサブヒープから構成される．それぞれのサブヒープは，リストにより表現され，ヒープメモリ全体は 3 つのリストの組として表現される．

各クラスのオブジェクト参照は，そのクラスに対応するサブヒープのインデックスで表現される．クラス `fig` に対応するサブヒープの場合，`fig` 型の参照 `f1`，`f2`，... は，それぞれ，自然数 1，2，... に対応している．`f0` は NULL 参照 `fig_null` として使用される．オブジェクトの実体は，各クラスのサブヒープに格納される

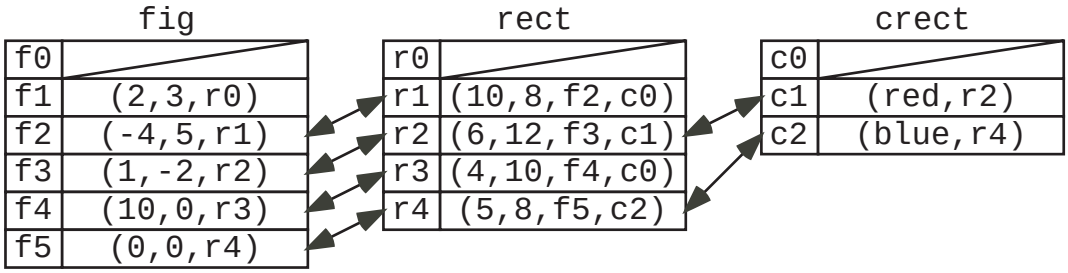


図 3.1: ヒープメモリの表現

複数のセルにより表現される．例えば，セル f1 は，属性値が x=2，y=3 である fig クラスのインスタンスを表す．2 つのセル f2，r1 は，属性値が x=-4，y=5，w=10，h=8 である rect クラスのインスタンスを表す．3 つのセル f3，r2，c1 は，属性値が x=1，y=-2，w=6，h=12，c=red である crect クラスのインスタンスを表す．1 つのオブジェクトを構成する複数のセルは，オブジェクト参照を互いに格納することによりリンクされる．1 つの rect インスタンスを構成する 2 つのセル f2，r1 は，参照 r1，f2 をそれぞれ格納することによりリンクしている．リンクするセルが存在しないときは，NULL 参照が格納される．セル f1 は，どの rect セルともリンクしないので，r0 を格納しているこのように，複数のセルから一つのオブジェクトを構成するメカニズムにより，オブジェクトのサブタイピングが実現される．例えば，3 つの参照 f3，r2，c1 はいずれも同じ crect インスタンスを指しているが，これは，このインスタンスが 3 つの見かけの型をとることができることを意味する．

次に，6 つの基本演算子の実装について説明する．new 演算子は，各クラスのメモリに新しいセルを追加し，それらを互いにリンクする関数として実装される．例えば，rect_new は，fig クラスのサブヒープ，rect クラスのサブヒープにそれぞれ 1 つづつセルを追加し，それらを互いにリンクする関数である．セルを追加する際，セルの各要素にはデフォルト値を格納する．ex 演算子は，参照がサブヒープの有効な範囲を指しているかを検査する述語として実装される．例えば，fig_ex は，fig クラスに対応するサブヒープにおいて，第一引数の fig 参照が，0 でなく，さらに，サブヒープの長さ以上でないことを検査する述語である．cast 演算子は，参照から，親クラスまたは子クラスの参照へセルを辿る関数として実装される．例

えば，fig_cast_rect は，第一引数の fig 参照が指すセルを取得し，それに格納される rect 参照を取得する関数である．get 演算子，set 演算子は，参照が指すセルに格納される属性を，それぞれ取得，更新する関数として実装される．is 演算子は，参照が指すセルのリンク構造に基づき，それがどのクラスのインスタンスであるか判別する述語として実装される．セルのリンク構造に着目すれば，そのクラスを一意に判別することができる．例えば fig 参照の場合，それが，f1 のように fig クラスのインスタンスを指しているならば，fig セルから rect セルへのリンクは NULL である．また，f2 のように rect クラスのインスタンスを指しているならば，fig セルから rect セルへのリンクは存在するが，rect セルから crect セルへのリンクは NULL である．f3 のように crect クラスのインスタンスを指しているならば，fig セルから rect セルへのリンク，rect セルから crect セルへのリンク両方が存在する．このように，セルのリンクがサブクラスの方向にどこまで辿れるかに注目すれば，fig 参照がどのクラスのインスタンスを指しているか識別可能である．is 演算子は，このようなリンク構造に基づいて実装される．

すべての公理はこれらの関数，述語の定義から導出可能である．例えば，公理 26 や 27 は関数定義の展開と自然数やリストに関する定理による書き換えのみによって簡単に証明することができる．公理 14 や 16 は特殊であり，ヒープメモリ構造の不変表明として証明される．不変表明の証明には，ヒープメモリに対する構造帰納法を用いる．つまり，初期段階として，ヒープメモリの初期値について成立することを証明し，帰納段階として，ヒープメモリに対する書き込み演算とオブジェクト生成演算の適用前後で成立することを証明する．

3.2 ヒープメモリの実装

3.2.1 サブヒープのデータ構造

サブヒープは，クラスモデルに依存せず，一般的にリスト $'a\ list$ として表現される．データのアドレスはリストのインデックス，つまり，自然数 0, 1, 2,... で表現する．サブヒープの初期値を $[null]$ とする．これは，アドレス 0 に NULL データを表すダミーの定数 $null : 'a$ を格納したリストである．サブヒープ上の演算子

として，$add, valid, read, write$ の 4 つを図 3.2 に定義する．

$$
\begin{array}{l}
add : 'a \to 'a\ list \to 'a\ list \\
add\ x\ l \equiv (Length\ l, Append\ l\ [x]) \\
\\
valid : num \to 'a\ list \to bool \\
valid\ n\ l \equiv (0 < n) \wedge (n < length\ l) \\
\\
read : num \to 'a\ list \to 'a \\
read\ n\ l \equiv if\ valid\ n\ l\ then\ read_1\ n\ l\ else\ unknown \\
\quad where \\
\quad\quad (read_1\ 0\ l = Hd\ l) \wedge (read_1\ (Suc\ n)\ l = read_1\ n\ (Tl\ l)) \\
\\
write : num \to 'a \to 'a\ list \to 'a\ list \\
write\ n\ x\ l \equiv if\ valid\ n\ l\ then\ write_1\ n\ x\ l\ else\ l \\
\quad where \\
\quad\quad (write_1\ 0\ x\ l = x :: (Tl\ l)) \wedge \\
\quad\quad (write_1\ (Suc\ n)\ x\ l = (Hd\ l) :: (write_1\ n\ x\ (Tl\ l)))
\end{array}
$$

図 3.2: サブヒープ上の演算子

add は，リストの最後尾にデータ x を追加し，そのアドレスと追加後のリストを返す関数である．$valid$ は，指定されたアドレス n に生成されたデータが存在するかを検査する述語である．アドレスが有効である範囲は，0 より大きく，現在のリスト長より小さい範囲である．$read$ は，指定されたアドレス n のデータを読みだす関数である．アドレスが無効である場合は，$unknown$ という未定義の値を意味する定数を返す．$write$ は，指定されたアドレス n にデータ x を書き込む関数である．アドレスが無効である場合は，ヒープに変更を加えない．

3.2.2 オブジェクト参照の表現

オブジェクト参照の型は，サブヒープのインデックス，つまり，自然数との間に全単射をとることにより得られる．クラス c のオブジェクト参照の型は次のように定義される．

$$HOL\_datatype\ c = AbsObj_c\ of\ num$$
$$RepObj_c\ (AbsObj_c\ n) \equiv n$$

関数 $AbsObj_c$ は自然数から c オブジェクトへの写像，関数 $RepObj_c$ は c オブジェクトから自然数への写像である．

NULL オブジェクトは 0 により表現する．つまり，

$$Null^c \equiv AbsObj_c\ 0$$

3.2.3 ヒープメモリのデータ構造

サブヒープはクラスモデルに存在する各クラスに対応して導入され，それぞれ異なる型のタプルが格納される．クラス c に対応するサブヒープに格納されるタプルの型を次のように定義する．

$$tuple_c \equiv attrs_c * d * e_1 * ... * e_m \quad (d \lhd c, c \lhd e_j)$$
$$where$$
$$attrs_c \equiv \mathcal{T}(c, a_1) * ... * \mathcal{T}(c, a_n) \quad (a_i \in \mathcal{M}_{attr}(c))$$

タプルの最初の要素 $attrs_c$ はクラス c で定義される属性をタプルでまとめたものである．次の要素 d は親クラスのオブジェクト参照であり，最後の m 個の要素 $e_1, ..., e_m$ は子クラスのオブジェクト参照である．オブジェクト参照を格納することによりタプル同士の連結を行う．このようなタプルを格納するクラス c のサブヒープの型を次のように定義する．

$$heap_c \equiv tuple_c\ list$$

ヒープメモリ全体は各クラスに導入されたサブヒープをタプルで結合すること

により得られる．ヒープメモリの型を次のように定義する．

$$Heap \equiv heap_{c_1} * \ ... \ * heap_{c_n} \ \ (c_i \in C)$$

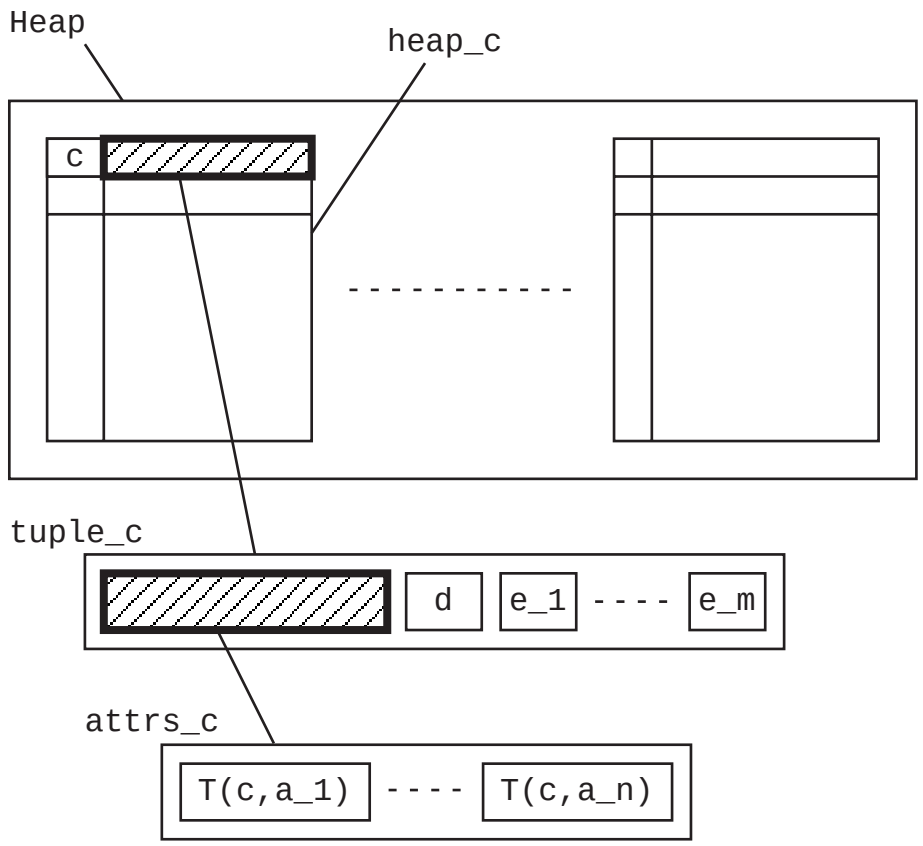


図 3.3: ヒープメモリのデータ構造

ヒープメモリに対する操作として，4 つの演算子 $Add^c, Valid^c, Read^c_u, Write^c_u$ を図 3.4 に定義する．$Valid^c$ はオブジェクト参照がクラス c に対応するサブヒープにおいて有効であるかどうかを検査する述語である．Add^c はクラス c に対応するサブヒープに新しいタプルを追加する関数である．$Read^c_u, Write^c_u$ はオブジェクト参照が指すタプルの要素 u に対し，それぞれ読み出し，書き込みを行う関数である．ただし，$u \in \{a, d, e_1, ..., e_m\}$ であり，$u = a$ のときは属性集合に対する読み書きであり $T = attr_c$，$u = d$ のときは親クラス参照に対する読み書きであり $T = d \ (d \lhd c)$，$u = e_j$ のときは子クラス参照に対する読み書きであり $T = e_j \ (c \lhd e_j)$ である．

ここで，$getElem^p_q$, $setElem^p_q$ は，q 個の要素を持つタプルの p 番目の要素を取

$$
\begin{array}{l}
Add^c : tuple^c \to Heap \to c * Heap \\
Add^c\ x\ H \equiv \\
\quad let\ (n, l) = add\ x\ (getElem^i_N\ H)\ in \\
\qquad (AbsObj^c\ n, setElem^i_N\ l\ H) \\
\\
Valid^c : c \to Heap \to bool \\
Valid^c\ o\ H \equiv valid\ (RepObj^c\ o)\ (getElem^i_N\ H) \\
\\
Read^c_u : c \to Heap \to T \\
Read^c_u\ o\ H \equiv \\
\quad if\ Valid^c\ o\ H\ then\ getElem^j_M\ (Read^c\ o\ H)\ else\ (unknown : T) \\
\ \ where \\
\quad Read^c\ o\ H \equiv (read\ (RepObj^c\ o)\ (getElem^i_N\ H)) \\
\\
Write^c_u : c \to T \to Heap \to Heap \\
Write^c_u\ o\ x\ H \equiv \\
\quad if\ Valid^c\ o\ H\ then \\
\qquad let\ t = setElem^j_M\ x\ (Read^c\ (RepObj^c\ o)\ H)\ in \\
\qquad\quad Write^c\ (RepObj^c\ o)\ t\ H \\
\quad else\ H \\
\ \ where \\
\quad Write^c\ o\ t\ H \equiv \\
\qquad let\ l = write\ (RepObj^c\ o)\ t\ (getElem^i_N\ H)\ in \\
\qquad\quad setElem^i_N\ l\ H
\end{array}
$$

図 3.4: ヒープメモリ上の演算子

得，更新する関数である．M はクラス集合 C の要素数であり，i はクラス c に対応するサブヒープの，ヒープメモリにおける位置である．N はクラス c に対応するサブヒープに格納されるタプルの要素数であり，j はそのタプルにおける u の位置である．$getElem^p_q$, $setElem^p_q$ は，p, q に依存して次のように定義される．

$$getElem^p_q\ t = \begin{cases} t & (p=1, q=1 \text{ のとき}) \\ Fst\ t & (p=1, 1<q \text{ のとき}) \\ Fst\ (Snd_{(1)}\ (...(Snd_{(p-1)}\ t))) & (1<p<q \text{ のとき}) \\ Snd_{(1)}\ (...(Snd_{(q-1)}\ t)) & (p=q, 1<q \text{ のとき}) \end{cases}$$

$$setElem^p_q\ x\ t = \begin{cases} x & (q=1 \text{ のとき}) \\ (getElem^1_q\ t, ..., getElem^{p-1}_q\ t, & \\ \quad x, getElem^{p+1}_q\ t, ..., getElem^q_q\ t) & (1<q \text{ のとき}) \end{cases}$$

3.2.4 ヒープメモリにおけるオブジェクト指向演算子の表現

ヒープメモリ上に定義した演算子を用いて，オブジェクト指向理論の定数 Emp, 演算子 $Ex^c, Cast^c_d, Get^c, Set^c, New^c, Is^c_d$ の $Heap$ における表現 $EmpRep, CastRep^c_d, GetRep^c, SetRep^c, NewRep^c, IsRep^c_d$ を定義する．ここで，Get^c, Set^c は，クラス c に定義される複数の属性を同時に取得，更新する関数であるとする．つまり，クラス c は複数の属性をまとめて 1 つのタプルとした属性を唯一持つと考え，Get^c, Set^c はそれを取得，更新する関数である．Get^c_a, Set^c_a はこれらの関数を用いて定義可能である．このような関数を導入するのは，後でヒープメモリのサブセットをストアに抽象する際に，そのサブセットを定義する述語における帰納段階を減らすことができ，証明の効率があがるからである．

$EmpRep$ は次のように定義される．

$$EmpRep : Heap$$
$$EmpRep \equiv ([null : tuple_{c_1}], ..., [null : tuple_{c_n}])\ \ (c_i \in C)$$

空のストアは，空のサブヒープをタプルにまとめたものとして表現される．

$ExRep^c$ は次のように定義される．

$$ExRep^c : c \to Heap \to bool$$
$$ExRep^c\ o\ H \equiv Valid^c\ o\ H$$

これは $Valid^c$ そのものである．つまり，クラス c のオブジェクトがストアに存在することはその参照がクラス c に対応するサブヒープにおいて有効であること同値である．

$GetRep^c$, $SetRep^c$ はそれぞれ次のように定義される．

$$GetRep^c : c \to H \to attr_c$$
$$GetRep^c\ o\ H \equiv Read_a^c\ o\ H$$

$$SetRep^c : c \to attr_c \to H \to H$$
$$SetRep^c\ o\ x\ H \equiv Write_a^c\ o\ x\ H$$

クラス c のオブジェクトの属性の取得，更新は，そのオブジェクト参照が指すタプルの属性領域の取得，更新と同値である．

$CastRep_d^c$ は次のように定義される．

$$CastRep_d^c : c \to Heap \to d$$
$$CastRep_d^c\ o\ H \equiv$$
$$\begin{cases} if\ ExRep^c\ o\ H\ then\ Read_d^c\ o\ H\ else\ Null^d & (c \lhd d \text{ or } d \lhd c) \\ CastRep_d^e\ (CastRep_e^c\ o\ H)\ H & ((c \lhd e, e \lhd^+ d) \text{ or } (d \lhd^+ e, e \lhd c)) \end{cases}$$

$CastRep_d^c$ はクラス c と d の継承関係の位置について 2 つの場合に分けて定義される．変換元のクラス c と変換先のクラス d が直接の親子関係にある場合は，$Read_d^c$ により，オブジェクト参照が指すブロックに格納されるクラス d のオブジェクト参照を読み出せばよい．変換を行うオブジェクトが存在しない場合は変換先のクラスの NULL オブジェクト $Null^d$ に変換したこととする．c と d が直接の親子関係ではない継承関係にある場合は，型変換を推移的に適用すればよい．つまり，まず c と直接親子関係にあるクラス e に変換してから次にそれを d に変換する．

$NewRep^c$ は次のように定義される．

$$
NewRep^c\ H \equiv \begin{cases} Add^c\ default_c\ H & (c\text{がルートのとき}) \\ let\ (o_1, H_1) = NewRep^d\ H\ in & \\ \quad let\ (o_2, H_2) = Add^c\ default_c\ H_1\ in & \\ \quad\quad let\ H_3 = Link_c^d\ o_1\ o_2\ H_2\ in\ \ (o_2, H_3) & (d \lhd c) \end{cases}
$$

$$
where
$$

$$
Link_c^d\ o_1\ o_2\ H \equiv Write_d^c\ o_2\ o_1\ (Write_c^d\ o_1\ o_2\ H)
$$

$$
default_c \equiv ((\mathcal{V}(c, a_1), ..., \mathcal{V}(c, a_n)), Null^d, Null^{e_1}, ..., Null^{e_m})
$$

$$
(a_i \in \mathcal{M}_{attr}(c), d \lhd c, c \lhd e_j)
$$

$NewRep^c$ はクラス c のオブジェクトを，ルートクラスからクラス c に至るまでの継承関係に関して再帰的に構築する．生成されたオブジェクトは複数のタプルが連結した構造となる．まず，初期段階としてルートクラスのインスタンスを生成する場合，Add_c により，クラス c のサブヒープに新たなタプルを追加する．次に，帰納段階としてルート以外のクラスのインスタンスを生成する場合，まず，$NewRep^d$ により親クラス d のインスタンスを生成する．次に，Add^c により，クラス c のサブヒープに新たなタプルを追加し，最後に，得られた二つのオブジェクト参照 o_1, o_2 を $Link_d^c$ によりリンクする．$Link_d^c$ は，$Write_c^d$, $Write_d^c$ によって，o_1, o_2 それぞれが指すタプルのオブジェクト参照の要素に o_2, o_1 を書き込む関数として定義される．なお，クラス c のサブヒープに追加するタプル $default_c$ の各要素は，属性 a の場合はそのデフォルト値のタプル $(\mathcal{V}(c, a_1), ..., \mathcal{V}(c, a_n))$，親クラス d，子クラス e_j のオブジェクト参照の場合はそのクラスの NULL オブジェクト $Null^d$, $Null^{e_j}$ とする．

$IsRep_d^c$ は次のように定義される．

$$
IsRep_d^c\ o\ H \equiv
$$

$$
\begin{cases} ExRep^c\ o\ H \wedge \bigwedge_j \neg ExRep^{e_j}\ (CastRep_{e_j}^c\ o\ H)\ H & (c = d, c \lhd e_j) \\ ExRep^c\ o\ H \wedge IsRep_d^e\ (CastRep_e^c\ o\ H)\ H & (c \lhd e, e \lhd^* d) \end{cases}
$$

$IsRep_d^c$ は見かけの型 c を持つオブジェクトがクラス d のインスタンスであるかど

うかを判定する述語であるが，これを判定するには，型 c のオブジェクト参照が指すタプルから祖先クラスの方向にクラス d のサブヒープに格納されるタプルまでリンクしていることを調べればよい．子クラスのタプルとリンクしているかどうかは，子クラスに型変換した結果のオブジェクトがストアに存在するかで判別することができる．この述語は $c = d$ と $c \lhd^+ d$ の 2 つの場合に分けて定義される．$c = d$ である場合，c オブジェクトからどの子クラス e_j のタプルにもリンクしていなければよい．つまり，c オブジェクトを子クラス e_j に型変換した結果のオブジェクトがストアに存在しなければよい．$c \lhd^+ d$ である場合，c オブジェクト参照を d を子孫とする子クラス e に型変換し，それが d のインスタンスであればよい．いずれの場合も c オブジェクトがストアに存在していなければならない．

3.3 オブジェクト指向理論の導出

以上のように構築したヒープメモリをオブジェクト指向理論に抽象化する．具体的には，型 $store$ の生成，オブジェクト指向演算子の定義，及び，公理の導出を行う．

3.3.1 ストア型の生成

型 $store$ は，型 $Heap$ の部分集合から生成される．HOL において，既存の型 t_1 から新しい型 t_2 を生成するためには次のステップを踏む．

1. t_2 を表現する t_1 の部分集合を定める特徴述語 $p : t_1 \rightarrow bool$ を定義する．
2. この集合が少なくとも一つ要素を持つこと，つまり，$\exists x.\ p\ x$ を証明する．
3. t_1 と，p によって定義される t_2 の部分集合の間に全単射が存在することを表明する．

まず，$Heap$ 型の部分集合を定める特徴述語 $IsStoreRep$ を次のように定義する．

$$
\begin{array}{l}
IsStoreRep\ H \equiv \forall P.\ IsInv\ P \Rightarrow P\ H \\
\quad where \\
\quad\quad IsInv\ P \equiv P\ EmpRep \wedge \\
\quad\quad\quad \bigwedge_c (\forall o\ x\ H.\ P\ H \Rightarrow P\ (SetRep^c\ x\ H)) \wedge \\
\quad\quad\quad \bigwedge_c (\forall H.\ P\ H \Rightarrow P\ (Snd\ (NewRep^c\ H)))
\end{array}
$$

$IsStoreRep$によって定義される$Heap$の部分集合は，$Heap$の要素のうち，次の帰納法により証明される不変表明Pを満たすものの集合である．

- 初期段階：$EmpRep$がPを満たすことを証明する．
- 帰納段階：Pがある$Heap$の要素について成り立つと仮定し，それに$SetRep^c$または$NewRep^c$を適用して得られる要素についてもPが成り立つことを証明する（合計$2|C|$ステップ）．

言い換えれば，$IsStoreRep$によって定義される$Heap$の部分集合は，$EmpRep$，及び，$EmpRep$に$SetRep^c$または$NewRep^c$を任意回適用して得られる要素である．このように定義するのは，属性更新またはオブジェクト生成を通してのみしかストアの状態を変更できないようにするためである．

次に，この部分集合に少なくとも一つ要素が存在することを証明する．これは次の定理として証明される．

$$
th \equiv \vdash IsStoreRep\ EmpRep
$$

最後に，$store$と$Heap$の部分集合の間に全単射が存在することを表明する．これはHOLが提供するML関数$new_type_definition(store, th)$を呼び出すことにより自動的に表明される．この全単射をそれぞれ次のように定義する．

$$
\begin{array}{l}
RepStore : store \rightarrow Heap \\
AbsStore : Heap \rightarrow store
\end{array}
$$

以上の手続きにより，$store$型が生成される．図3.5はヒープメモリとストアの対応を図示したものである．

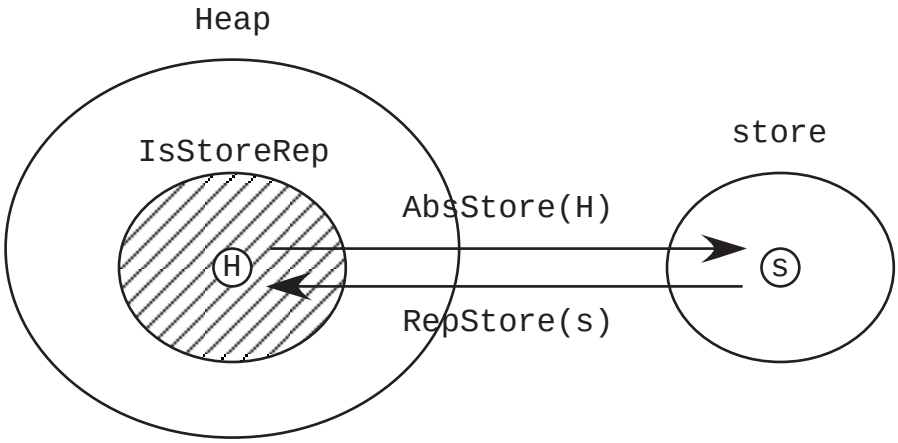


図 3.5: ヒープメモリからストアへの抽象化

3.3.2 演算子の定義

オブジェクト指向理論における定数，演算子は，それらのヒープにおける具体表現と関連付けることにより定義される．これらは全単射 $RepStore$ と $AbsStore$ を用いて次のように定義される．

$$
\begin{aligned}
&Emp \equiv AbsStore\ EmpRep \\
&Ex^c\ o\ s \equiv ExRep^c\ o\ (RepStore\ s) \\
&Get^c\ o\ s \equiv GetRep^c\ o\ (RepStore\ s) \\
&Set^c\ o\ x\ s \equiv AbsStore\ (SetRep^c\ o\ x\ (RepStore\ s)) \\
&Cast^c_d\ o\ s \equiv CastRep^c_d\ o\ (RepStore\ s) \\
&Is^c_d\ o\ s \equiv IsRep^c_d\ o\ (RepStore\ s) \\
&New^c\ s \equiv let\ (o, H) = NewRep^c\ (RepStore\ s)\ in\ (o, AbsStore\ H)
\end{aligned}
$$

Get^c_a, Set^c_a は Get^c, Set^c を使って次のように定義される．

$$
Get^c_a\ o\ s \equiv
\begin{cases}
if\ Ex^c\ o\ s\ then\ getElem^i_N\ (Get^c\ o\ s) & \\
\quad else\ Unknown^c_a & (a \in \mathcal{M}_{attr}(c)) \\
Get^d_a\ (Cast^c_d\ o\ s)\ s & (d \lhd c, a \in attr(d))
\end{cases}
$$

$$
Set^c_a\ o\ x\ s \equiv
\begin{cases}
if\ Ex^c\ o\ s\ then & \\
\quad let\ t = setElem^i_N\ x\ (Get^c\ o\ s)\ in & \\
\quad\quad Set^c\ o\ t\ s & \\
\quad else\ s & (a \in \mathcal{M}_{attr}(c)) \\
Set^d_a\ (Cast^c_d\ o\ s)\ x\ s & (d \lhd c, a \in attr(d))
\end{cases}
$$

Get^c_a は，属性 a がクラス c 自身で定義されたものか，または，祖先のクラスの属性を継承したものか，の 2 つの場合に分けて定義される．a が c で定義されている場合，Get^c により c オブジェクトの属性タプルを取得し，$getElem^i_N$ によりそのタプルの a の要素を取得する．ここで，N は c で定義される属性集合の要素数で，i はその属性集合における a の位置である．c オブジェクトが存在しない場合はその属性の未定義の値を意味する定数 $Unknown^c_a$ を返す．a が祖先クラス d で定義されている場合，c オブジェクトを d に型変換し，Get^c_a を適用する．

Set^c_a も同様に 2 つの場合に分けて定義される．a が c で定義されている場合，Get^c により c オブジェクトの属性タプルを取得し，$setElem^i_N$ によりそのタプルの a の要素を更新する．更新されたタプルを Set^c によりストアに反映する．c オブジェクトが存在しない場合はストアに変更を加えない．a が祖先クラス d で定義されている場合，c オブジェクトを d に型変換し，Set^c_a を適用する．

3.3.3 公理の導出

オブジェクト指向理論に含まれる公理は以上の定義から導出することができる．各公理の導出過程はパターン化しており，それぞれのパターン応じてカスタマイズされたタクティックにより自動的に証明される．

公理はその導出法により 2 種類に分類される．一つは定義から直接導出できるものであり，もう一つはヒープメモリ上の不変表明として導出できるものである．後者は，公理 3, 7, 14, 15, 16, 17 である．不変表明は $IsInv$ を満たす述語であり，その証明はヒープメモリの構造に基づく帰納法となっている．

不変表明の導出法を公理 6 を例にとって説明する．この公理を次の述語 Inv として定義し，$\vdash \forall s.\ Invs$ を証明する．

$$Inv\ s \equiv \forall o.\ Ex^c\ o\ s = Is^c_{d_1}\ o\ s \vee ... \vee Is^c_{d_n}\ o\ s\ \ (d_i \in \{d \mid c \lhd^* d\})$$

まず，この述語のヒープメモリにおける表現 $InvRep$ を次のように定義する．

$$InvRep\ H \equiv \forall o.\ ExRep^c\ o\ H = IsRep^c_{d_1}\ o\ H \vee ... \vee IsRep^c_{d_n}\ o\ H$$

次に，$InvRep$ がヒープメモリにおける不変表明であること，つまり，

$$\vdash IsInv\ InvRep$$

を証明する．この証明は，$IsInv$ を展開した後，帰納法の以下の各ステップ（合計 $1 + 2|C|$ 個）を証明すればよい．

$$\vdash InvRep\ EmpRep$$

$$\vdash \forall o\ x\ H.\ InvRep\ H \Rightarrow InvRep\ (SetRep^c\ x\ H))\ \ (c \in C)$$

$$\vdash \forall H.\ InvRep\ H \Rightarrow InvRep\ (Snd\ (NewRep^c\ H))\ \ (c \in C)$$

次に，この定理と $IsStoreRep$ の定義を用いて，

$$\vdash \forall H.\ IsStoreRep\ H \Rightarrow InvRep\ H$$

を証明する．これは，H がストアを表現する要素ならば，不変表明 $InvRep$ を満たすことを意味する．

また，ストアとヒープメモリの間の全単射を定義した際に自動的に証明される定理から次の定理が導出できる．

$$\vdash \forall s.\ IsStoreRep\ (RepStore\ s)$$

これら 2 つの定理から次の定理を導出する．

$$\vdash \forall s.\ InvRep\ (RepStore\ s)$$

つまり，ストア s のヒープメモリにおける表現は不変表明 $InvRep$ を満たす．

最後に，この定理と演算子 Ex^c, Is^c_d の定義を用いて，求める定理

$$\vdash \forall s.\ Inv\ s$$

が得られる．

その他の不変表明も同様に証明することができる．

第 4章

コラボレーションに基づく不変表明の検証

本章では，オブジェクト指向理論の応用として，コラボレーションに基づく不変表明の検証法を述べる．不変表明の検証法としては，状態遷移図に基づくものが提案されている [25][26] が，これらの手法と比較して，コラボレーションベースの手法は複数のオブジェクトにまたがる不変表明の検証に有効である．

本章の構成は以下の通りである．まず，4.1 節において既存の状態遷移図ベースの検証法とその問題点を述べる．次に，4.2 節において本論文で提案するコラボレーションベースの検証法を述べ，4.3 節において状態遷移図ベースの手法では扱いにくい性質の証明を行う．4.4 節では，アプリケーションの検証例として図書館システムの検証を行う．最後に，4.5 において考察を行う．

4.1 状態遷移図ベースの検証法

状態遷移図に基づく不変表明の検証法と，その問題点について述べる．

青木ら [25] は，状態遷移モデルに基づき，個々のクラスに与えられた不変表明を証明するための公理系を定義している．このモデルは，オブジェクトの振る舞いを，UML におけるステートチャート図のような状態遷移に基づき，形式的に定義している．この手法では，まず，検証対象のクラスに対し，証明したい不変表明を割り当てる（大域表明と呼ぶ）．この表明を証明するために，各状態に，その

状態で成立することが期待される表明を割り当て（局所表明と呼ぶ），それらが常に成立することを状態遷移図に関する構造帰納法により証明する．これら証明された局所表明から最終的に目的の大域表明を導出する．

図 4.1 は，counter の状態遷移モデルである．ここで，状態遷移に付加される E1/A/E2 の形式は,「イベント E1 を受信したとき，アクション A を行って状態遷移し，遷移後，イベント E2 を送信する」を意味する．A や E2 がないときは省略して E1//E2 や E1/A/のように書く．この counter は属性として，整数型の number を持つ．number の値はどのような状態遷移が起こったとしても，常に 0 以上をとる．

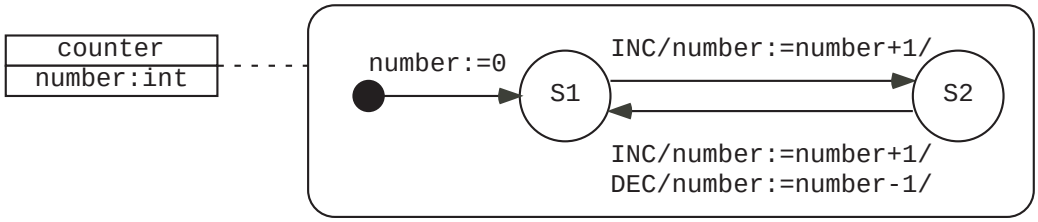


図 4.1: counter の状態遷移図

この性質を証明するにあたって，まず，大域表明 GA を以下のように定義する．

```
GA(x) = (0 <= x)
```

次に，状態 S1，S2 について，局所表明 LA1，LA2 をそれぞれ以下のように割り当てる．

```
LA1(x) = (0 <= x)
LA2(x) = (1 <= x)
```

これらの表明がその割り当てられた状態で常に成立することは，次の帰納法により証明される．

```
Base step:
|- LA1(0)
Induction steps:
|- !x. LA1(x) ==> LA2(x+1)
|- !x. LA2(x) ==> LA1(x+1)
|- !x. LA2(x) ==> LA1(x-1)
```

初期段階は，初期状態の表明 LA1 を number の初期値 0 が満たすことを意味する．３つの帰納段階はいずれも，現在の状態について表明が成立することを仮定し，遷移先の状態でその表明が更新された属性値について成立することを意味する．LA1, LA2 はそれぞれ GA を含意するので，GA がこの counter のいかなる状態においても成立することが証明される．

この手法により，個々のオブジェクト単独の振る舞いを完全に捉えることができるが，一般には，オブジェクトの振る舞いは，互いに協調動作する別のオブジェクトの振る舞いに影響される．このような場合，個々のオブジェクトの性質はその状態遷移図に着目するだけでは証明することができない．図 4.2 は相互作用する person と counter の状態遷移図を示す．この counter は属性として，整数型の number を持つ．その値は person から送信されるイベント INC，DEC に呼応して，それぞれ，加算，減算される．person は，加算用ボタン，減算用ボタンを交互に押すことによってそれら２つのイベントを交互に送信する．この場合，counter は任意の整数値を持つことができるにもかかわらず，その値は，0 または 1 に制限されてしまう．counter に関するこの事実は，person の振る舞いを考慮しなければ分からない．

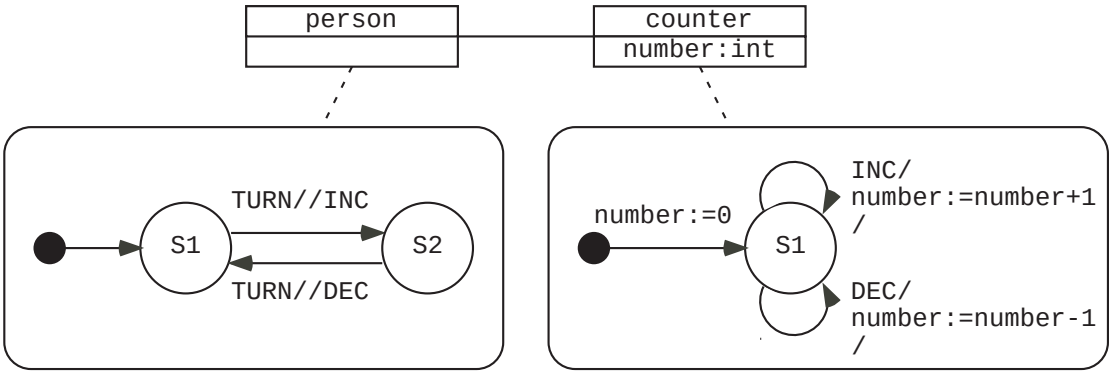


図 4.2: 協調動作する person と counter の状態遷移図

この場合に対処する最も単純なアプローチは，すべてのオブジェクトの状態遷移図の直積をとり一つの状態遷移図にまとめることである．しかし，この方法は状態爆発の問題により，一般に適用できない．

立石ら [26] は，このような複数のオブジェクトにまたがる不変表明を，状態を合

成することなく証明するための手法として，段階的振る舞い近似法を提案している．この手法では，各クラスの状態遷移図が TSE (Transition Sequence Expression) と呼ばれる，正規表現に似た式に変換される．各 TSE は，イベント送受信相手の TSE の振る舞いに従って段階的に近似される．この段階的近似は，目的の不変表明が証明されるまで継続される．証明はホーア論理に似た体系で行われる．つまり，TSE に対し前条件，後条件を割り当て，いくつかの推論規則により，それが成立することを導出する．この手法を適用すると，図 4.2 の状態遷移図は，図 4.3 の状態遷移図に近似される．person から送信されるイベント列は INC，DEC の繰り返しであるから，counter の状態遷移図は，それに応じて交互に状態遷移するものへと近似される．この近似された状態遷移図からであれば，number の値が 0 または 1 であることを容易に推論することができる．

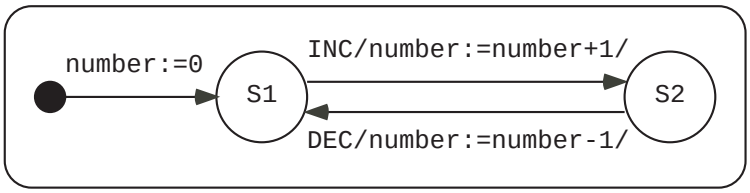


図 4.3: 近似された counter の状態遷移図

この手法では，近似の途中段階でも証明を行うことが可能であるため，状態を合成する方法よりも効率的である．それでも，近似を行う作業には大きなコストを要してしまう．状態遷移図ベースの検証法により複数のオブジェクトにまたがる性質を証明する場合，こういった，複数のオブジェクトの状態遷移図からそれらの相互作用に関する情報を抜き出すコストが常に必要となる．本研究では，このような性質をクラス構造を横断するコラボレーションの視点から検証することを考える．

4.2 コラボレーションベースの検証法

コラボレーションはシステムのまとまった機能を実現するオブジェクト間の協調動作である．例えば，図書館業務管理システムにおいては貸出手続きや返却手続きに相当する．コラボレーションベースの検証法の基本的な考え方は，複数の

オブジェクト間にまたがるコラボレーションを一本の関数適用列と考え，システム全体をコラボレーションの集合として構成することである．不変表明はシステムの初期状態からの遷移列に関する帰納法により証明する．

4.2.1 システムの構成と不変表明の証明法

一般に，システムには複数のコラボレーションが存在し，それらが繰り返し適用されることによりシステムは状態変化する．すなわち，システムは次の二つ組で定義される．

$$System = (S, F)$$

S はシステムのとりうる状態の集合であり，初期状態 s_0 を含む．F はコラボレーションの有限集合であり，各コラボレーションは状態を変化させる関数 $f : S \to S$ である．f は $f : int \to S \to string * S$ のように入出力を伴うことがあるが，ここでは簡単のため入出力を省略する．

コラボレーションの並行性は考慮しない．つまり，それぞれのコラボレーションはアトミックに適用され，インタリーブすることはない．

システムが不変表明を満たすことを証明するためには初期状態からの遷移列に関する帰納法を用いる．つまり，不変表明 P を S 上の述語としたとき，

- 初期段階：$\vdash P(s_0)$
- 帰納段階：$\vdash \forall s.P(s) \Rightarrow P(f(s))$ $(f \in F)$

の証明を行う．

4.2.2 検証の概要

検証の流れを図 4.4 に示す．まず，ユーザは対象システムのクラスモデルを定義する．クラスモデルは特定の形式でファイルに宣言され，理論生成器に入力される．理論生成器とは，我々が HOL のメタ言語である Moscow ML [3] で実装したツールであり，対象システムのクラスモデルに基づき，そのシステムに特化した

HOL 理論を構築する．HOL における理論とは，型，定数，公理，定理を格納する ML モジュールである．ユーザはこの理論に与えられる基本演算子と HOL が提供する関数群を用いて，コラボレーション，及び，不変表明をそれぞれ，関数，述語として定義する．最後に，不変表明を，システムの初期状態からの遷移列に関する帰納法により証明する．

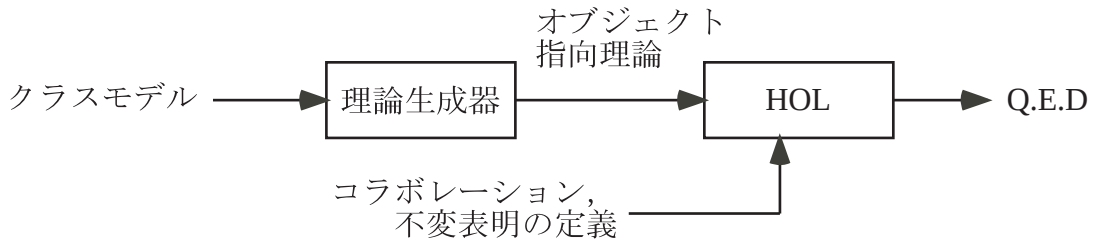


図 4.4: 検証の流れ

4.3 検証例

4.1 節では，複数のオブジェクトにまたがる不変表明の証明は，状態遷移図ベースの検証法では困難であることを述べた．本節では，そのような性質をコラボレーションベースで検証する方法を説明する．例題として，4.1 節の person と counter の例を用いる．検証は 4.2 節の流れに従う．

4.3.1 クラスモデルの定義

クラスモデルを図 4.5 に示す．クラス person は，next 型の属性 next を持つ．型 next は，次の列挙型として定義される．

```
next = inc | dec
```

この属性は，person オブジェクトが counter に対し次に行う操作を定めるものである．つまり，next の値が inc ならば，次に加算操作を行うことを意味する．また，dec ならば，減算操作を行う．もう一つの属性として，counter 型の属性 counter を持つ．この属性は，person オブジェクトが操作する counter オブジェ

クトを保持するためのものである．counter クラスは，int 型の属性 number を持つ．number にはカウント数が保持される．

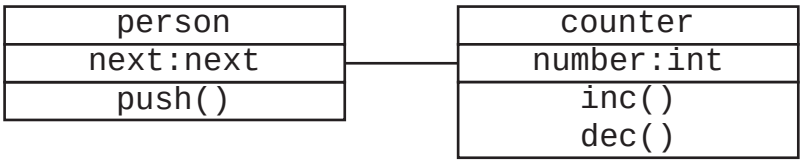


図 4.5: person と counter のクラスモデル

4.3.2 初期状態の定義

クラスモデルは理論生成器に入力され，そのモデルに特化したオブジェクト指向理論が生成される．この理論で定義される演算子を用いて，まず，システムの初期状態を定義する．

counter クラスのオブジェクト生成メソッドを new_counter として次のように定義する．

```
new_counter : store -> counter # store
new_counter s =
   let (c,s1) = counter_new s in
   let s2 = counter_set_number c 0 s1 in
      (c,s2)
```

new_counterは，まず，new 演算子 counter_newを用いて新しい counterオブジェクトを生成し，次にそのオブジェクトに対し，set 演算子 counter_set_number を用いて属性 number の初期値を 0 にセットする．出力として，新たに生成したオブジェクトの参照と，生成後のストアを返す．

同様に person クラスのオブジェクト生成メソッドを次のように定義する．

```
new_person : counter -> store -> store
new_person c s =
   let (p,s1) = person_new s in
```

```
   let s2 = person_set_next p inc s1 in
   let s3 = person_set_counter p c s2 in
      (p,s2)
```

new_personは，新しいpersonオブジェクトを生成し，属性nextの初期値をincにセットする．つまり，personオブジェクトは最初に加算操作を行うこととなる．また，引数で与えられるcounterオブジェクトを属性counterの値にセットし，リンクを張る．

これら2つのクラスのオブジェクト生成メソッドを用いて，システム全体の初期化関数sys_initを次のように定義する．

```
sys_init : store -> person # store
sys_init s =
   let (c,s1) = new_counter s in
   let (p,s2) = new_person c s1 in
      (p,s2)
```

この関数はストアに適用され，2つのクラスからオブジェクトを生成し，personオブジェクトと生成後のストアを出力する．この関数をストアの初期値store_empに適用することにより，システムの初期状態が生成される．システムの初期状態，生成されるpersonオブジェクトをそれぞれ定数S0，PERSONとして次のように定義する．

```
S0 = SND (sys_init store_emp)
PERSON = FST (sys_init store_emp)
```

このようにして定義されるシステムの初期状態を図4.6に示す．

4.3.3 コラボレーションの定義

次に，personオブジェクトとcounterオブジェクトのインタラクションをコラボレーションとして定義する．

2つのオブジェクトの振る舞いはシーケンス図として図4.7のように記述される．ガード条件の分岐が2つ存在するため2つの図に分けて記述している．コラボレー

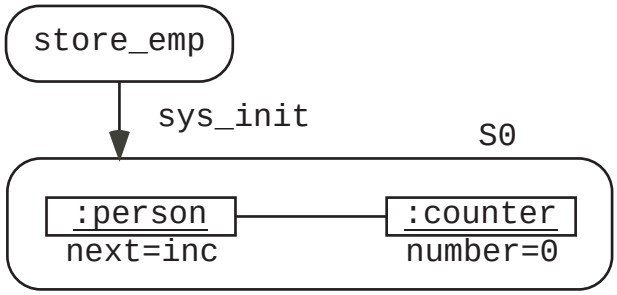


図 4.6: システムの初期状態

ションの起点となるのは person オブジェクトのメソッド push() の呼び出しである．メソッド push() は，属性 next の値が inc ならば counter オブジェクトのメソッド inc() を呼び出し，dec ならば dec() を呼び出す．メソッド inc(), dec() は属性 number の値をそれぞれ加算，減算する．メソッド呼び出し後は，next の値を反転させる．つまり，inc() の呼び出し後は dec に，dec() の呼び出し後は inc に変更する．

メソッド push(), inc(), dec() は HOL においてそれぞれ以下の関数person_push, counter_inc, counter_dec として定義される．

```
person_push : person -> store -> store
person_push p s =
   let c = person_get_counter p s in
      if person_get_next p s = inc then
         let s1 = counter_inc c s in
            person_set_next p dec s1
      else
         let s1 = counter_dec c s in
            person_set_next p inc s1

counter_inc : counter -> store -> store
counter_inc c s =
   let x = counter_get_number c s in
      counter_set_number c (x+1) s
```

```
counter_dec : counter -> store -> store
counter_dec c s =
   let x = counter_get_number c s in
      counter_set_number c (x-1) s
```

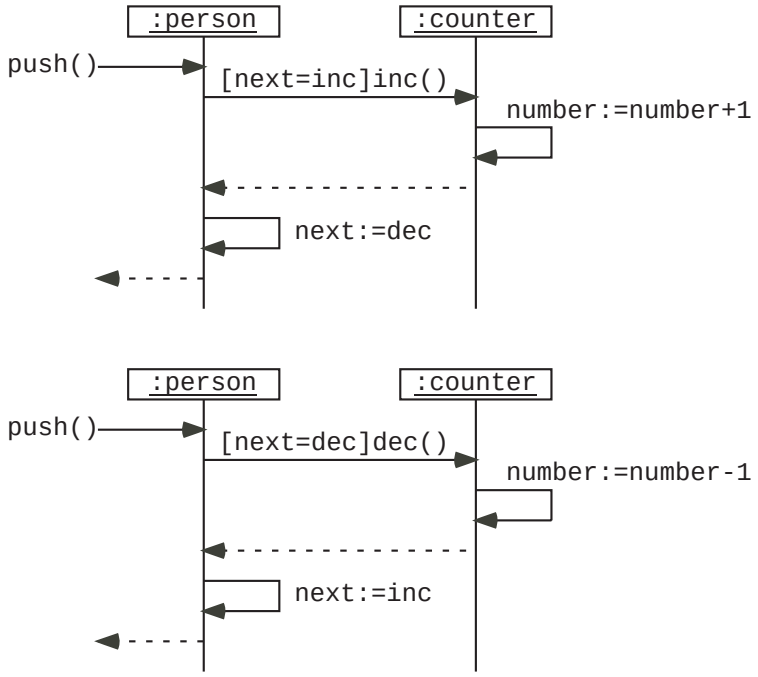


図 4.7: コラボレーションのシーケンス図

このように定義されるコラボレーションの適用前後で変化するシステムの状態を図 4.8 に示す.

4.3.4 不変表明の定義

不変表明はストア上の述語として定義する．証明対象となる不変表明の OCL (Object Constraint Language [4]) による記述は次の通りである.

person

```
0 <= counter.number <= 1
```

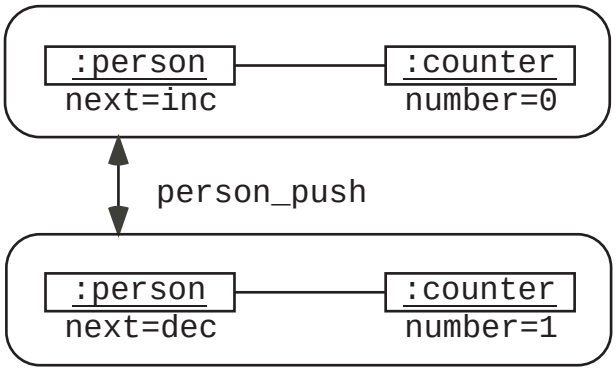


図 4.8: コラボレーション前後のシステムの状態変化

つまり，person オブジェクトを文脈（context）とし，それとリンクする counter オブジェクトの属性 number の値が常に 0 以上 1 以下である，という性質である．ここで，文脈とは OCL における概念であり，不変表明を記述する視点となるオブジェクトを表す．文脈としては，システム中の任意のクラスのオブジェクトを選ぶことが可能であるが，ここで，person オブジェクトを文脈としているのは，コラボレーションの始点となるオブジェクトと一致し，証明が単純化するためである．

HOL において，この不変表明は次のように定義される．

```
Inv1 : person -> store -> bool
Inv1 p s = person_ex p s ==>
   let c = person_get_counter p s in
   let x = counter_get_number c s in
      (0 <= x) /\ (x <= 1)
```

Inv1 は，不変表明の文脈である person オブジェクトを引数とする．この person オブジェクトがストアに存在している必要があるので，person_ex p s を仮定している．

不変表明が成立するためには，しばしば，他の不変表明の成立が前提として必要となる．Inv1 の証明には，次の二つの不変表明が前提として必要である．

```
Inv2 : person -> store -> bool
Inv2 p s = person_ex p s ==>
  let c = person_get_counter p s in
```

```
    counter_ex c s

Inv3 : person -> store -> bool
Inv3 p s = person_ex p s ==>
   let c = person_get_counter p s in
      if person_get_next p s = inc then
         (counter_get_number c s = 0)
      else
         (counter_get_number c s = 1)
```

Inv2は，personオブジェクトがストアに存在するならば，それとリンクするcounterオブジェクトもストアに存在することを意味する．Inv3は，personオブジェクトが次に加算操作を行うときは，counterオブジェクトのnumberの値は0であり，減算を行うときは1であることをを意味する．

これら3つの不変表明をまとめて，システム全体の不変表明を次のように定義する．

```
Inv : person -> store -> bool
Inv p s = Inv1 p s /\ Inv2 p s /\ Inv3 p s
```

4.3.5 不変表明の証明

不変表明の証明には，システムの初期状態からの遷移列に関する帰納法を適用する．このシステムの初期状態はS0であり，唯一のコラボレーションとしてperson_pushを持つ．したがって，次の帰納法により，Invがシステムのすべての状態で成立することが証明される．

```
Base step:
|- Inv PERSON S0
Induction step:
|- !p s. Inv p s ==> Inv p (person_push p s)
```

InvはInv1を含意するので，目的の不変表明が証明される．

4.4 アプリケーションの検証例

アプリケーションの検証例として図書館システムの検証を行った．この図書館システムは，図書館における利用者の登録や本の登録，貸出，返却手続きといった業務を支援するシステムである．

4.4.1 クラスモデルの定義

クラスモデルを図 4.9 に示す．library クラスはこのシステム全体を管理するクラスであり，登録や，貸出などのコラボレーションの始点となるメソッドを持つ．属性 max, days はそれぞれ，利用者の最大貸出数，最大貸出日数である．属性 nextcid は利用者の登録の際，次に発行される利用者 ID である．同様に, nextiid は物品の登録の際，次に発行される物品 ID である．

customer クラスは図書館に登録される利用者のクラスである．属性 cid, name はそれぞれ，利用者 ID，利用者の名前である．同じ名前の利用者は同一の利用者としてみなす．

item クラスは図書館に登録される物品のクラスである．属性 iid, title は物品 ID，物品のタイトルである．物品には，本と CD の 2 種類があり，それぞれ item クラスのサブクラス book, cd とする．book クラスは ISBN を表す属性 isbn を持つ．

lend クラスは customer クラスと item クラスの間に関連を持ち，対応する利用者と物品の貸出情報を保持する．貸出情報として，残り貸出日数を意味する属性 days を持つ．この値が負になると貸出が延滞されていることになる．関連は，型がオブジェクトのリストである属性として表現する．例えば，library クラスから customer クラスへの関連は，library クラスの属性 customerlist : customer list として表現する．ここでリスト型とするのは，関連の多重度が 0 以上であるためである．集合型を用いることもできるが，リスト上の高階関数 MAP, FILTER などがメソッド定義の際有用であるためリスト型を選択している．その他の関連も同様に定義する．

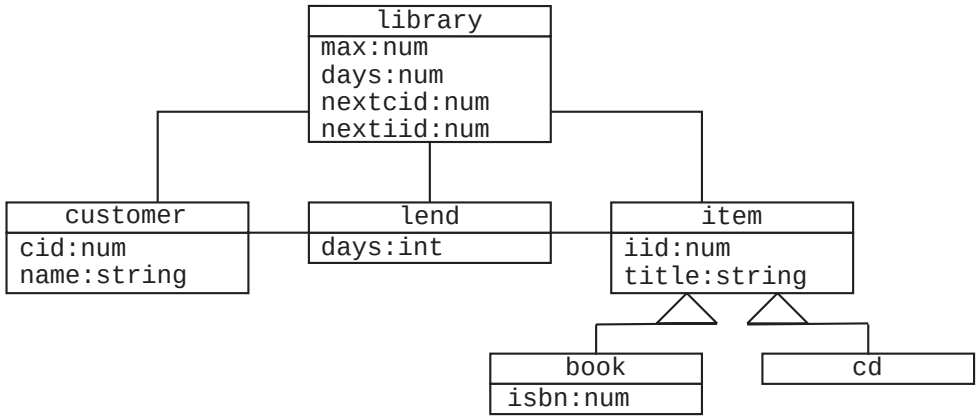


図 4.9: 図書館システムのクラスモデル

4.4.2 コラボレーションの定義

図書館システムには，利用者登録，本登録，CD 登録，貸出，返却，日付更新の 6 つのコラボレーションが存在する．

貸出のコラボレーション図を図 4.10 に示す．このコラボレーションは，library クラスのメソッド lend() が始点となる．まず，利用者 ID と物品 ID を入力とし，対応する利用者と物品が貸出可能な状態にあるかチェックする (1.1)．チェックする条件は，利用者 ID，物品 ID が有効であること，利用者の現在の貸出数が，規定される最大貸出数未満であること，利用者が貸出を延滞している物品を保持していないこと，物品がどの利用者にも貸し出されていないことである．この条件が満たされるならば，ID に対応する customer オブジェクト，item オブジェクトを library オブジェクトに接続するリンク集合から取得し (1.2, 1.3)，規定の最大貸出日数を取得する (1.4)．次に，lend オブジェクトを生成し，残り貸出日数を最大貸出日数にセットする (1.5.1)．そして，この lend オブジェクトと，customer オブジェクト，item オブジェクトをそれぞれ互いにリンクする (1.5.2-1.5.5)．最後に，生成した lend オブジェクトを library オブジェクトにリンクする (1.6)．

貸出コラボレーションの HOL による定義を図 4.11, 4.12 に示す．関数定義と図 4.10 との対応は関数脇の番号によって示す．現時点では，コラボレーション図から HOL 関数への形式的な変換法は定義していないが，基本的には，メソッドをオブジェクト指向演算子を用いた関数として定義し，コラボレーション全体はそれらの適用列

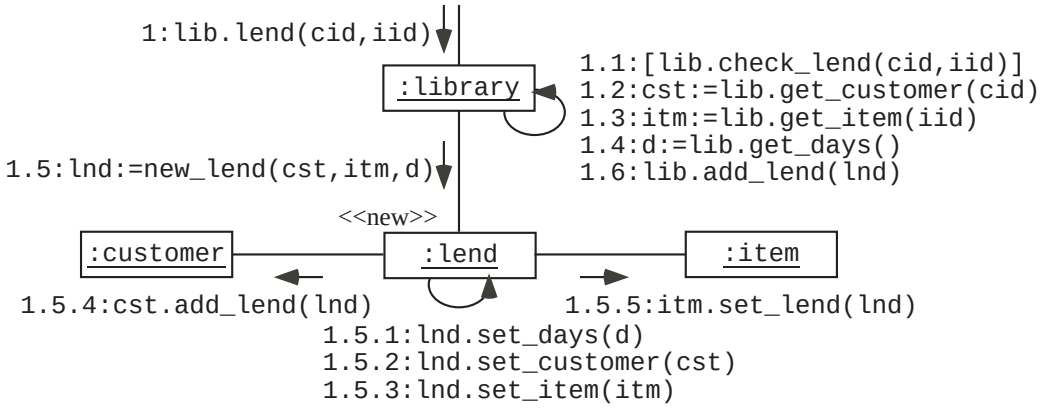


図 4.10: 貸出手続きのコラボレーション

として定義している．例えば，1.2におけるメソッドは関数library_get_customerとして定義している．このメソッドは，入力のcidと同じIDを持つcustomerオブジェクトを返す関数であるが，これは，get演算子library_get_customerlistによりlibraryオブジェクトにリンクしているcustomerオブジェクトのリストを取得し，リスト関数HD，FILTER，及びget演算子customer_get_cidを用いてcidと一致するオブジェクトを一つだけ取り出すという関数として実装している．1.5におけるメソッドはlendオブジェクトの生成メソッドであり，関数new_lendとして定義される．この内部の1.5.4におけるメソッドは関数customer_add_lendとして定義している．この関数では，get演算子customer_get_lendlistとset演算子customer_set_lendlist，及びリストの::(cons)を用いて，生成されたlendオブジェクトlndをリストに追加している．

その他のコラボレーションは次の型を持つ関数として定義される．

- 利用者登録：名前を入力し，登録成功・非成功メッセージと発行した利用者IDを出力する．

```
library_register_customer :
  library -> string -> string # num # store
```

- 本登録：タイトル，ISBNを入力し，発行した物品IDを出力する．

```
library_register_book : library -> string -> num -> num # store
```

```
library_lend : library -> num -> num -> store -> string # store
library_lend lib cid iid s =                          (* 1 *)
  if library_check_lend lib cid iid s then   (* 1.1 *)
    let cst = library_get_customer lib cid s in   (* 1.2 *)
    let itm = library_get_item lib iid s in       (* 1.3 *)
    let d = library_get_days lib s in             (* 1.4 *)
    let (lnd,s1) = new_lend d cst itm s in        (* 1.5 *)
    let s2 = library_add_lend lib lnd s1 in       (* 1.6 *)
      ( ok ,s2)
  else
      ( fail ,s)

new_lend : num -> customer -> item -> store -> lend # store
new_lend d cst itm s =
  let (lnd,s1) = lend_new s in
  let s2 = lend_set_days lnd d s1 in         (* 1.5.1 *)
  let s3 = lend_add_customer lnd cst s2 in   (* 1.5.2 *)
  let s4 = lend_add_item lnd itm s3 in       (* 1.5.3 *)
  let s5 = customer_add_lend cst lnd s4 in   (* 1.5.4 *)
  let s6 = item_add_lend itm lnd s5 in       (* 1.5.5 *)
    (lnd,s6)

library_check_lend : library -> num -> num -> store -> bool
library_check_lend lib cid iid s =
  if library_check_cid lib cid s /\
     library_check_iid lib iid s then
    let cst = library_get_customer lib cid s in
    let itm = library_get_item lib iid s in
    customer_get_lendnum cst s < library_get_max lib s /\
    customer_check_over cst s /\
    item_is_available itm s
  else
    false

library_check_cid : library -> num -> store -> bool
library_check_cid lib cid s =
  let l = library_get_customerlist lib s in
    MEM cid (MAP (\x. customer_get_cid x s) l)

library_check_iid : library -> num -> store -> bool
library_check_iid lib iid s =
  let l = library_get_itemlist lib s in
    MEM iid (MAP (\x. item_get_iid x s) l)
```

図 4.11: HOL における貸出コラボレーションの定義

```
library_get_customer : library -> num -> store -> customer
library_get_customer lib cid s =
  let l = library_get_customerlist lib s in
    HD (FILTER (\x. customer_get_cid x s = cid) l)

library_get_item : library -> num -> store -> item
library_get_item lib iid s =
  let l = library_get_itemlist lib s in
    HD (FILTER (\x. item_get_iid x s = iid) l)

customer_get_lendnum : customer -> store -> num
customer_get_lendnum cst s = LENGTH (customer_get_lendlist cst s)

customer_check_over : customer -> store -> bool
customer_check_over cst s =
  let l1 = customer_get_lendlist cst s in
    EVERY (\x. 0 <= lend_get_days x s) l1

item_is_available : item -> store -> bool
item_is_available itm s = (LENGTH (item_get_lendlist itm s) = 0)

customer_add_lend : customer -> lend -> store -> store
customer_add_lend cst lnd s =
  let l = customer_get_lendlist cst s in
    customer_set_lendlist cst (lnd::l) s

library_add_lend : library -> lend -> store -> store
library_add_lend lib lnd s =
  let l = library_get_lendlist lib s in
    library_set_lendlist lib (lnd::l) s
```

図 4.12: HOL における貸出コラボレーションの定義（続き）

- CD 登録：タイトルを入力し，発行した物品 ID を出力する．

  ```
  library_register_cd : library -> string -> num # store
  ```

- 返却：物品 ID を入力し，返却成功・非成功メッセージを出力する．

  ```
  library_return : library -> num -> store -> string # store
  ```

- 日付更新：library オブジェクトはリンクしているすべての lend オブジェクトの属性 days を 1 減算する．

  ```
  library_clock : library -> store -> store
  ```

4.4.3 不変表明の定義と証明

このシステムついて成立すべき不変表明「全利用者の貸出数の総和は，貸出中の物品の総数に等しい」の証明を行った．この不変表明は次の OCL 式で定義される．

```
library
customer.get_lendnum()->sum() =
item->select(not(is_available()))->size
```

ここで，get_lendnum()，is_available() は利用者の貸出数を取得する関数，物品が貸出可能であるかチェックする述語であるとする．

HOL では Inv1 として図 4.13 のように定義される．不変表明の左辺において，OCL ナビゲーション演算 customer.get_lendnum() はリスト関数 MAP により実現している．つまり，get 演算子 library_get_customerlist によって取得される customer オブジェクトのリスト個々の要素に対しメソッド customer_get_lendnum を適用し，貸出数のリストに変換する．また，右辺において，OCL 集合演算 select, size はそれぞれリスト関数 FILTER, LENGTH により実現している．customer_get_lendnum, item_is_available の定義は図 4.12 に含まれている．

Inv1 を証明するためには，いくつかの不変表明が前提として必要である．例えば,「library オブジェクトにリンクしている customer オブジェクトはストアに存

```
Inv1 : library -> store -> bool
Inv1 lib s = library_ex lib s ==>
  (library_get_customer_lendsum lib s = library_get_item_lendsum lib s)

library_get_customer_lendsum : library -> store -> num
library_get_customer_lendsum lib s =
  let l = library_get_customerlist lib s in
    SUM (MAP (\x. customer_get_lendnum x s) l)

library_get_item_lendsum : library -> store -> num
library_get_item_lendsum lib s =
  let l = library_get_itemlist lib s in
    LENGTH (FILTER (\x. ~(item_is_available x s)) l)
```

図 4.13: HOL における不変表明の定義

在し，さらに同じ customer オブジェクトが重複してリンクしていない」などがある．これらをまとめて Inv2 として定義する．

Inv1 が常に成り立つことは，以下の帰納法により証明される．

```
Base step:

|- Inv1 LIBRARY S0

Induction steps

|- !lib s. Inv1 lib s /\ Inv2 lib s ==>

   Inv1 lib (SND (SND (library_register_customer lib name s)))

|- !lib s. Inv1 lib s /\ Inv2 lib s ==>

   Inv1 lib (SND (library_register_book lib title isbn s))

|- !lib s. Inv1 lib s /\ Inv2 lib s ==>

   Inv1 lib (SND (library_register_cd lib title s))

|- !lib s. Inv1 lib s /\ Inv2 lib s ==>

   Inv1 lib (SND (library_lend lib cid iid s))

|- !lib s. Inv1 lib s /\ Inv2 lib s ==>

   Inv1 lib (SND (library_return lib iid s))

|- !lib s. Inv1 lib s /\ Inv2 lib s ==>

   Inv1 lib (library_clock lib s)
```

ここで，LIBRARY はシステム初期化時に生成される library オブジェクト，S0 はシステムの初期状態である．6 つの帰納段階はシステムに存在する 6 つのコラボ

レーションに対応する．

Inv1 の成立に本質的に関係するコラボレーションは，lend オブジェクトへのリンク接続操作を行う貸出，及び返却手続きであり，これらに対応する 2 つの帰納段階の証明には計 8 時間程度を要した．初期段階との残りの帰納段階については，ほぼ自動的に証明することができた．

その他，以下の不変表明が証明可能である．

- 利用者の貸出数は規定の最大貸出数を超えない．

  ```
  library
  customer->forall(cst | cst.lend<=self.max)
  ```

- 物品は同時に 2 人以上の利用者に貸し出されていない．

  ```
  library
  item->forall(itm | itm.lend<=1)
  ```

- 登録される物品に割り当てられる物品 ID はそれぞれ異なっている．

  ```
  library
  item->forall(itm1,itm2 | itm1<>itm2 implies itm1.iid<>itm2.iid)
  ```

4.5 考察

4.5.1 検証効率について

状態遷移図ベースの検証法と，本論文で提案するコラボレーションベースの検証法の決定的な違いは，オブジェクト数に関する検証のスケーラビリティである．状態遷移図ベースの検証法においては，複数のオブジェクトにまたがる性質を証明する際は，状態遷移図の合成をとる必要がある．合成により状態爆発が起こってしまった場合は検証すること自体が不可能となる．段階的振る舞い近似法のような状態数を削減する手法を用いたとしても，その効率はオブジェクト数の増加に伴い急激に悪化する．一方，コラボレーションベースの検証法では最初から合

成システムの無限の状態を一つの変数として扱っており，任意の数のオブジェクトに対応することが可能である．そのため，オブジェクト数の増加が効率悪化に直接つながることはなく，検証自体不可能となることもない．オブジェクト数の増加に対応できることは，多くのオブジェクトにより構成される大規模システムを検証する上で重要である．また，コラボレーションベースの検証法では，メソッド呼び出しというオブジェクト間の協調動作を直接捉える視点でシステムの振る舞いを定義できるため，段階的振る舞い近似法のように，合成された状態遷移図から協調動作に関わる状態遷移図を抜き出すオーバーヘッドもない．

ただし，状態を一変数で扱うことによるデメリットもある．それは，コラボレーション，つまり，一変数上に適用される関数適用列を一括して証明対象としなければならず，ユーザにかかる証明の負担が大きくなることである．4.1で述べた青木らの手法では，複数の状態を考慮する分，個々の状態で実行されるアクションは単純化し，各状態遷移に対応する帰納段階の証明はほぼ自動的に行うことができる．

証明の複雑さは定理証明の大きな問題であるが，それに対応するためには，高級なタクティックの実装，及び，証明の整理法の研究が必要である．タクティックとは，HOLのGoal-Oriented Proofにおける分解規則であり，MLによって既存のタクティックからより強力なタクティックを実装することができる．この機能を利用してオブジェクト指向理論に特化した高級タクティックを実装することにより，証明ステップを削減することができると考えられる．証明の整理法としては，メソッドを単位として補題を作成する方法が有効であると考えられる．これは，複数のコラボレーションが同一のメソッドを共有していることがあるためであり，そのメソッドに関して補題を蓄積しておけば，複数の証明に再利用することができる．この点を考慮して，証明方法論を明確にする必要がある．

4.5.2 コラボレーションの計算モデルについて

本論文では，コラボレーションをシステムの状態の上の関数として定義しているが，現段階でその具体的な計算モデルまでは定義していない．コラボレーションをHOLに実装する際は，高階述語論理の範囲で任意のものが定義できてしまう

のが現状である．開発者にとって理解しやすい高級な計算モデルを定義し，HOL 表現との対応付けを定義する必要がある．コラボレーションの定義法に制限が与えられれば，その定義の構造に基づいた，より詳細な証明方論法が明確になる．現時点では，コラボレーションをシーケンシャルなオブジェクト指向プログラムとして定義し，ホーア論理に基づく検証条件自動生成ツールを実装することを考えている．なお，コラボレーションの意味論に関する研究としては，UML シーケンス図をイベントトレースの集合として解釈する方法 [23] や，UML コラボレーション図をグラフ変換規則として解釈する方法 [24] がある．

4.5.3 並行性について

person と counter の例題は，person オブジェクトが能動的なオブジェクトであり，counter オブジェクトは受動的なオブジェクトであった．言い換えれば，動作の始点になるのは常に person オブジェクトであり，counter オブジェクトは person オブジェクトによるメソッド呼び出しがあってはじめて動作する．このようなコラボレーションはシーケンシャルな実行の流れであり，関数適用列として容易に表現することができる．しかし，実際に存在するシステムの中には，分散システムのように，複数の能動オブジェクトが存在し，それらが自立的に動作することにより構成されるものが数多く存在する．そのようなシステムについての不変表明を証明するためには，並行性の概念を扱えるようにする必要がある．

第 5 章

検証支援ツール

本章では，HOL ライブラリ objLib として実装したオブジェクト指向理論の検証ツールについて述べる．ライブラリは以下の 2 つの機能を提供する．

- オブジェクト指向理論の自動生成
- オブジェクト指向理論のシミュレーション実行環境の自動生成

それぞれ，ML 関数 mk_theory_script, mk_theory_simurator として実装される（図 5.1）．

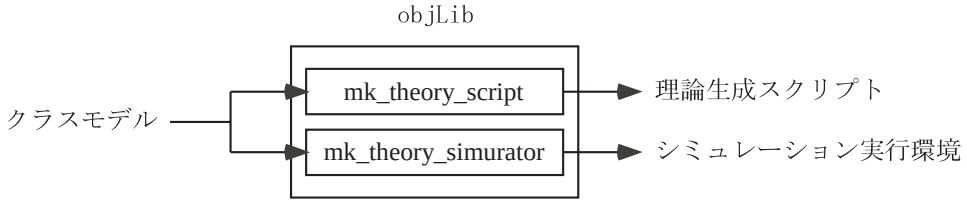


図 5.1: objLib の機能

まず，両関数の共通の入力となるクラスモデルについて述べる．次に，理論生成機能，実行環境生成機能それぞれについて述べる．

5.1 クラスモデル

クラスモデルは特定の形式に従ってファイルに記述される．図 5.2 は図 2.1 の例のクラスモデルを記述したファイル figure.cm である．

```
class fig
attr x  holtype : int | 0   mltype : int | 0
attr y  holtype : int | 0   mltype : int | 0

class rect extends fig
attr w  holtype : num | 0   mltype : int | 0
attr h  holtype : num | 0   mltype : int | 0

class crect extends rect
attr c  holtype : color | black  mltype : color | black
```

図 5.2: クラスモデルファイルの例

クラスの宣言は，

```
class C extends S
```

の形式によって行う．これは C がクラス S を継承するクラスであることを表す．

属性の宣言は，

```
attr X  holtype : HT | HD   mltype : MT | MD
```

の形式によって行う．これは，名前が X，HOL における型とデフォルト値がそれぞれ HT, HD，ML における型とデフォルト値がそれぞれ MT, MD であることを表す．HOL における型とはオブジェクト指向理論における型であり，ML における型とはシミュレータにより ML 上で実行する際の型である．属性 w, h の宣言において，num 型は ML には存在しないため，int で代用している．

5.2 オブジェクト指向理論の自動生成

5.2.1 理論スクリプトファイルの自動生成

HOL において理論を生成するためには，理論スクリプトファイルと呼ばれる，理論を生成するための HOL スクリプトが記述されたファイルを用意し，それを Holmake という HOL が提供するシェル用コマンドに通す必要がある．ML 関数 objLib.mk_theory_script は，クラスモデルが定義されたファイルを入力し，それに特化したオブジェクト指向理論を構築するための理論スクリプトファイルを出力する．

例えば，クラスモデル figure.cm を mk_theory_scriptの入力として実行すると，figureScript.sml という理論スクリプトファイルが生成される．これを Holmake に通すことにより，オブジェクト指向理論 figureTheory.sml が得られる．この理論には store, fig などの型や fig_new, fig_get_x などの演算子が含まれている．

図 5.3: 理論の構築過程

5.2.2 公理の自動生成

生成される理論には公理は含まれていない．これは公理の総数が非常に多く（クラス数，属性数の二乗に比例して増加），すべてを予め証明しておくのは効率が悪いためである．代わりに全 41 種類の公理を証明するための 41 個の ML 関数をライブラリが提供する．ユーザは必要になればそれらを呼び出して公理を取得する．例えば，次の関数

```
objLib.GET_SET : {Get:term} -> thm
```

は，get 演算子を入力として，公理 26 を返す関数である．get 演算子を入力するのはそれが公理を一意に識別するキーとなるためである．この関数を使うためには

予めライブラリにクラスモデルをロードしておく必要がある．

```
> objLib.load{File="/.../figure.cm"};
- val it = () : unit
> objLib.GET_SET{Get=‘‘fig_get_x‘‘};
- val it = |- !o x s. fig_ex o s ==>
                (fig_get_x o (fig_set_x o x s) = x) : thm
```

それでも公理が要求されるごとに毎回自動証明を行うのは時間効率が悪いため，軽量版として，公理を証明せずに直接導入する方法で公理を取得する関数も実装した．公理の直接導入は ML 関数 mk_thm を用いて行うことができるが，理論に矛盾を含める危険性があり，避けるべきである．しかし，他の定理証明器で証明された理論を HOL に移植する場合などには推奨されている方法である．本研究では，アプリケーションに特化したオブジェクト指向理論のように，自動生成しなければ構築できないような理論を効率的に生成するための手段として用いることも正当な使用法であると考えている．勿論，軽量版での証明完了後にはそれに用いた公理を実際に証明することが健全性を保証する上で必要となる．

5.2.3 タクティック

証明の際，多くの公理を一つづつ細かく適用していくのは効率が悪い．そこで，ゴールの形に応じて自動的に公理を適用するタクティックを実装した．現在のところ，継承の演算子が関わらない範囲でのみ適用可能である．

主なタクティックは，SLICE_TAC，OBJ_SIMP_TAC の 2 つである．SLICE_TAC はスライサの役目を持つタクティックであり，Ex 演算子や Get 演算子の値に影響しない New 演算子や Set 演算子を除去するタクティックである．例えば，ゴールに fig_get_x f (fig_set_y f y s) という項が含まれているならば，公理 28 に基づき，fi_set_y の適用を除去し，fig_get_x f s に簡略化する．OBJ_SIMP_TAC は，スライサの適用によって残される本質的な部分の証明を試みるタクティックである．例えば，ゴールに fig_get_x f (fig_set_x f x s) という項が含まれているならば，公理 26 の適用を試みる．

上の 2 つのタクティックの働きを以下に例示する．証明の基本的な流れは,「定義の展開→ SLICE_TAC の適用→ OBJ_SIMP_TAC の適用」となる．次の命題は，図書館システムに関するものであり,「lend オブジェクトの生成後，customer オブジェクトの貸出数が 1 増加する」という命題である．関数 g は命題を証明のゴールに設定する．

```
- g ‘!cst s. customer_ex cst s ==>
     (customer_get_lendnum cst (SND (new_lend d cst itm s)) =
      customer_get_lendnum cst s + 1)‘;;
```

まず，関数定義を展開し，基本演算子が現れるようにする．REWRITE_TACは引数のリストに与えられる定理群で書き換えを行うタクティックであり，LET_PAIR_TAC は，let (x,y) = z in ... の項を展開するタクティックである．

```
- e (REWRITE_TAC [customer_get_lendnum,new_lend,customer_add_lend]
     THEN LET_PAIR_TAC);;
> val it =
    !cst s. customer_ex cst s ==>
      (LENGTH
         (customer_get_lendlist cst
            (item_set_lend itm (FST (lend_new s))
               (customer_set_lendlist cst
                  (FST (lend_new s)::
                       customer_get_lendlist cst
                         (lend_set_item (FST (lend_new s)) itm
                            (lend_set_customer (FST (lend_new s)) cst
                               (lend_set_days (FST (lend_new s)) d
                                  (SND (lend_new s))))))
                  (lend_set_item (FST (lend_new s)) itm
                     (lend_set_customer (FST (lend_new s)) cst
                        (lend_set_days (FST (lend_new s)) d
                           (SND (lend_new s)))))))) =
       LENGTH (customer_get_lendlist cst s) + 1) : goalstack
```

ここで，Get 演算子 customer_get_lendlist の適用に直接影響しない Set 演算子 item_set_lend，lend_set_item や New 演算子 lend_new が適用されている．こ

のような場合，SLICE_TAC を適用すれば，それら関係のない演算子を一掃することができる．

```
- e SLICE_TAC;;
> val it =
    !cst s. customer_ex cst s ==>
      (LENGTH
         (customer_get_lendlist cst
            (customer_set_lendlist cst
               (FST (lend_new s)::customer_get_lendlist cst s) s)) =
       LENGTH (customer_get_lendlist cst s) + 1 : goalstack
```

スライス後は，Get 演算子 customer_get_lendlist に直接影響する Set 演算子 customer_set_lendlistだけが残される．これらの演算子はOBJ_SIMP_TACによって簡単化することができる．

```
- e OBJ_SIMP_TAC;;
> val it =
    !cst s. customer_ex cst s ==>
      (LENGTH (FST (lend_new s)::customer_get_lendlist cst s) =
       LENGTH (customer_get_lendlist cst s) + 1) : goalstack
```

ここで OBJ_SIMP_TAC により適用されたのは次の公理 26 である．

```
!i x s . customer_ex i s ==>
    (customer_get_lendlist i (customer_set_lendlust i x s) = x)
```

最後に残ったのはリストに関する性質であるから，リストの定理群を用いたタクティックにより証明することができる．

```
- e (RW_TAC list_ss []);;
> val it =
    Initial goal proved.
    |- !cst s. customer_ex cst s ==>
         (customer_get_lendnum cst (SND (new_lend d cst itm s)) =
          customer_get_lendnum cst s + 1) : goalstack
```

図書館システムの検証では，個々の公理を単独で適用することは一切せずに，SLICE_TAC，OBJ_SIMP_TAC のみで証明することができた．

$Get^c_a\ o_1\ (Set^c_a\ o_2\ x\ s)$や$Get^c_a\ o\ (Snd\ (New^c\ s))$の形の項は基本的にOBJ_SIMP_TACによって簡単化されるが，しばしば，簡単化できずに残ることがある．このような場合，場合分けを行うタクティックGET_SET_CASES_TAC，GET_NEW_CASES_TACを適用すればよい．これらのタクティックは，その形の項をいくつかの場合分けて簡単化する．

GET_SET_CASES_TACは，項tが項$Get^c_a\ o_1\ (Set^c_a\ o_2\ x\ s)$を含むとき（ただし，$o_1, o_2, x, s$は束縛変数を含まない項），ゴール$\{u\}t$（$u$が仮定，$t$が証明対象の命題）を次の3つのサブゴールに分解する．

$$\{u,\ \neg(Ex^c\ o_1\ s)\}\ t\ [Unknown^c_a/Get^c_a\ o_1\ (Set^c_a\ o_2\ x\ s)]$$
$$\{u,\ \neg(o_1 = o_2)\}\ t\ [Get^c_a\ o\ s/Get^c_a\ o_1\ (Set^c_a\ o_2\ x\ s)]$$
$$\{u,\ Ex^c\ o_1\ s,\ o_1 = o_2\}\ t\ [x/Get^c_a\ o_1\ (Set^c_a\ o_2\ x\ s)]$$

以下はその適用例である．

```
- g ‘item_get_title itm1 (item_set_title itm2 x s) = y‘;
> ...
- e GET_SET_CASES_TAC;
> val it =
    item_unknown_title = y
    ------------------------------------
      ~item_ex itm1 s

    item_get_title itm1 s = y
    ------------------------------------
      ~(itm1 = itm2)

    x = y
    ------------------------------------
      0.  itm1 = itm2
      1.  item_ex itm1 s
      : goalstack
```

GET_NEW_CASES_TACは，項tが項$Get^c_a\ o\ (Snd\ (New^c\ s))$を含むとき（ただし，

o, s は束縛変数を含まない項），ゴール $\{u\}t$ を次の 3 つのサブゴールに分解する．

$$\{u,\ \neg(o = Fst\ (New^c\ s)),\ \neg(Ex^c\ o\ s)\}\ t\ [Unknown^c_a/Get^c_a\ o\ (Snd(New^c\ s))]$$

$$\{u,\ Ex^c\ o\ s\}\ t\ [Get^c_a\ o\ s/Get^c_a\ o\ (Snd(New^c\ s))]$$

$$\{u,\ o = Fst\ (New^c\ s)\}\ t\ [\mathcal{V}(c,a)/Get^c_a\ o\ (Snd(New^c\ s))]$$

以下はその適用例である．

```
- g ‘item_get_title itm (SND (item_new s)) = y‘;
> ...
- e GET_NEW_CASES_TAC;
> val it =
    item_unknown_title = y
    ------------------------------------
      0.  ~(itm = FST (item_new s))
      1.  ~item_ex itm s

    item_get_title itm s = y
    ------------------------------------
      item_ex itm s

    "" = y
    ------------------------------------
      itm = FST (item_new s)
      : goalstack
```

5.3 シミュレーション実行環境の自動生成

第 4 章で述べたように，ユーザは，オブジェクト指向理論が提供する演算子を用いてメソッドやコラボレーションを関数として定義し，定理証明を行うこととなる．HOL において定義される関数は当然実行不可能であるが，それらの動作を実行により予め確認することができれば検証の効率化につながる．つまり，コストの高い定理証明に先立ち，比較的コストの小さいシミュレーション実行によってできる限りバグを洗い出しておくことにより，検証プロセス全体でのコストを

削減することができる．objLibはそのような実行環境を生成する機能も持つ．

オブジェクト指向理論はヒープメモリの操作的意味論に基づいて構築されているが，その操作的意味論をMLにおける実際の操作として実装することにより，実行が可能となる．ML関数objLib.mk_theory_simulatorはクラスモデルを入力とし，実行可能なオブジェクト指向演算子を含むMLストラクチャを生成する．

例えば，クラスモデルfigure.cmからはfigure.smlというストラクチャが生成される．図5.4はこのストラクチャをオープンした後のシミュレーション実行の様子である．

```
> val (r,s1) = rect_new store_emp;
- val r = <rect> : rect
- val s1 = <store> : store
> val s2 = rect_set_x r 10 s1;
- val s2 = <store> : store
> val f = rect_cast_fig r s2;
- val f = <fig> : fig
> fig_is_rect f s2;
- val it = true : bool
> fig_get_x f s2;
- val it = 10 : int
```

図 5.4: シミュレーション実行の様子

まず，rect_newを空のストアstore_empに適用する．これによって得られるのはrectオブジェクトrと生成後のストアs1である．オブジェクトやストアの内部構造は不透明なシグネチャ制約により隠蔽している．次にrect_set_xにより，rの属性xの値を10にセットする．次にrect_cast_figによりrを親クラスfigに型変換し，fとする．このfigオブジェクトfはrectクラスのインスタンスであるから，fig_is_rectを適用した結果はtrueとなる．最後に，fig_get_xにより，fの属性xを取得する．この値は子クラスにおいてセットした値10となって

いる．

ユーザは，これらのオブジェクト指向演算子を用いて，MLプログラムとしてメソッドを定義し実行することが可能である．ただし，MLで定義される関数は必ずしも停止しないので注意する必要がある．HOLにおいては停止性が保証される関数のみしか定義することを許されない．また，MLとHOLの間でデータ型の意味の不一致がある可能性がある．つまり，同じ名前のデータ型とその上に定義される関数が必ずしも同じ動作をするとは限らない．このような場合，ML上での実行結果とHOLでの証明結果が異なる場合が起こりうる．現時点ではMLとHOLの対応関係についてはユーザの判断に任せられるが，対応関係を明確にした上で，MLとHOLの両方に変換するための共通言語を定義するということも考えられる．

第 6 章

関連研究

6.1 オブジェクト指向概念の高階述語論理への実装

6.1.1 ヒープメモリに基づくオブジェクト指向理論

Berg ら (LOOP[8]) は，JML 仕様の付加された Java プログラムの検証の意味論として，ヒープメモリ構造を Isabelle/HOL と PVS に一般的な理論として定義している．ヒープメモリの構成要素は型無しのブロックであり，任意の Java オブジェクトを格納することができる．型無しブロックは簡単に表現すれば，次のようなリストと直積を使った型として定義される．

```
int list # bool list # ref list
```

ここで，Java に出現する型は `int`, `bool`, `ref` の 3 つであるとする．`ref` はオブジェクト参照の型である．このブロックには一つのオブジェクトのすべての属性を格納することができる．例えば，`int` 型の属性 `x=2`，`y=3`，`bool` 型の属性 `b=T` を持つオブジェクトは

```
([2,3],[T],[])
```

のようにブロックに格納される．どの属性がどの位置に格納されるかという情報は別に与えられる．このような型無しプロックを定義すれば，任意の Java オブジェクトを格納できるヒープメモリを一般的に構築することができる．

このような定義が可能であるのは，ヒープメモリに格納するデータ型が，Java に出現する有限個に限られるからである．本研究は，プログラミング言語ではなく分析モデルを検証対象としているため，対象システムに出現する任意の型を扱える必要がある．任意の型を格納するためのメモリ構造を一般的な理論として簡潔な形で定義することは HOL の型制約では困難であるため，対象システムのクラスモデルからそれに特化した理論を自動生成するという手段を選んだ．

勿論，LOOP の理論においても，Java クラスとして任意のデータ型を表現することは可能である．しかし，それらのクラスから型の性質を導出するための証明が余分に必要となる．本研究の長所は，HOL のライブラリとして既にその性質が証明された既存の多種多様な型を直接オブジェクトの属性に組み込んで検証することができることである．また，新しい型が必要な場合も，その型の構築は HOL のプロセスとしてオブジェクトとは無関係に行うことができる．型に関する証明をオブジェクトに関する証明と独立して行うことができることは，関心事の分離という点で重要である．

6.1.2 Extensible record による構造的サブタイピング

Naraschewski ら [11] は，Isabelle/HOL において，オブジェクトを extensible record と呼ばれるデータ型により表現している．Extensible record は`{x=3,y=5,...}`のように表記されるデータ型であり,「...」の部分に新たな要素を追加することにより，`{x=3,y=5,f=true,...}`と拡張することができる．このデータ型は，最後の要素を型変数としたタプル `int#int#'a` で表現されており，`'a` を `bool#'a` で詳細化することにより拡張を行っている．継承は，サブクラスにおいてレコードの要素を追加していくことにより実現される．

このようなデータ型によりオブジェクトの構造的なサブタイピングが可能となり，オーバーライドや動的束縛といった概念を実現することができる．しかし，オブジェクトが参照としての役割を持たないため，協調動作を実現することは不可能である．協調動作が実現できることは，それをベースに機能が構成されるアプリケーションの検証にとって特に重要である．したがって，本研究では，ストアという概念を導入し，オブジェクトをそれに格納されるデータへの参照として表

現することにより，協調動作を可能とした．

6.1.3 ストアに基づくオブジェクト指向理論

A. Poetzsch-Heffer[12] らは，Java プログラム検証のためのホーア論理を定義している．ホーア論理の意味論としてストアに基づくオブジェクト指向理論を HOL に実装している．ストア上の演算子は，get, set, new, $alive$ の 4 つである．$alive$ は本研究の ex 演算子に相当する．この理論では，継承に関する演算子は定義されておらず，継承に関する公理はホーア論理のレベルで定義されている．本研究では cast 演算子と is 演算子を導入することにより，ストアレベルで継承を実現した．

6.2 オブジェクト指向モデル一貫性の検証

6.2.1 B 仕様記述に基づく UML モデルの検証

UML/OCL モデルの検証に関する研究には [15][16] がある．これらの手法では，UML と OCL で記述されたモデルの B 仕様記述への変換法を定義し，クラスに付加される不変表明や，オペレーションに付加される事前，事後表明が一貫していることを Ateiler-B などの定理証明器を用いて検証することが可能である．これらの手法では，オペレーションは事前，事後表明を付加することによってのみ定義されるだけであり，オペレーションの内部で行われる協調動作までは定義することができない．我々が提案するオブジェクト指向理論においては，オペレーション内部の協調動作を HOL の関数を用いて定義することが可能である．

6.2.2 公理的意味論に基づく UML 状態遷移図の検証

青木ら [25] は UML モデルの一貫性検証の公理系を HOL に定義している．この手法では，4.1 節でも述べたように，クラスに割り当てられる表明が常に成立することを状態遷移図の遷移列に基づく帰納法により証明することが可能である．公理系の構築に関しては，公理を直接導入する方法で行っており，その信頼性に問題がある．本論文では，この手法では難しい，複数のクラスにまたがる性質の検

証法として，関数ベースの手法を提案した．また，本論文で提案するオブジェクト指向理論は，HOL に既存の無矛盾な理論から健全な推論規則を用いて導出することにより構築しており，健全性が成り立つ．

6.3 コラボレーションに基づく設計手法のための検証

フィーチャー指向プログラミング [17] に関する研究が盛んになっている．オブジェクト指向方法論においてはオブジェクトを単位としてシステムを構成するのに対し，フィーチャー指向方法論では，システムが提供する機能（フィーチャーと呼ぶ）を単位としてそれを構成する．携帯端末などの組み込みソフトウェアに見られるように，最近のソフトウェアには，類似の仕様のもと機能が多様化するという特徴がある．フィーチャー指向方法論はそのようなソフトウェアを効率的に開発するためのプロダクトラインアーキテクチャとして有効であるといわれている [21][18]．フィーチャー指向に基づく一般的な設計手法は，まず，基底となるシステムを構築し，それに対し層状にコラボレーションを追加することにより全体を段階的に構築していくというものである．フィーチャーとコラボレーションは同義語的に扱われ，この構築法をコラボレーションに基づく設計法 (Collaboration-based design) という．

このような設計手法に対する検証手法として，モデル検査に基づくものが提案されている [20][19]．基本的な検証手法は，個々のコラボレーションの性質を独立に検証し，システムにコラボレーションを追加する際に，それが既存のシステムの性質を維持することを証明する，というものである．個々のコラボレーションは状態遷移図により定義され，CTL で記述される性質が検証可能である．定理証明器に基づく検証手法には，ACL2 を用いた事例研究がある [22]．ACL 式により記述されるコラボレーションの一階述語論理に関する性質が証明可能である．本論文で提案するオブジェクト指向理論はコラボレーションに基づく設計手法の基礎理論として応用できると考えている．ACL2 の表現能力は一階述語論理であり，データ型として整数，ブール値，リストしか持たないのに対し，HOL は，高階述語論理であり表現能力が高く，強力なデータ型構築機能，及び豊富なデータ型ライブラリ群を持つためより柔軟なモデル構築が可能である．

第 7章

おわりに

本論文では，分析モデル検証のためのオブジェクト指向理論を HOL に実装し，コラボレーションに基づいて不変表明を証明する手法を提案した．

オブジェクト指向理論は，属性が任意の型をとることを可能とするため，アプリケーションのクラスモデルに特化して自動生成される．これにより，分析モデルに出現する様々な型に関する検証が可能となる．また，理論は definitional Extension により構築され，健全性が保証される．具体的には，オブジェクトを格納するためのヒープメモリ構造を定義し，その操作的意味論から公理系を導出している．さらに，ヒープメモリの操作的意味論を ML の実際の操作として実装することにより理論のシミュレーション実行を可能とした．

コラボレーションベースの検証法では，コラボレーションをオブジェクト指向理論により与えられる基本的な演算子を用いた関数適用列として定義し，不変表明はシステムの初期状態からの遷移列に関する帰納法により証明する．この手法は，状態遷移図ベースの検証法と比較して，複数のオブジェクトにまたがる不変表明の検証に有効である．

今後の課題は，コラボレーションの計算モデルを形式化し，その証明方法論を確立すること，また，オブジェクト指向理論をコラボレーションに基づく設計手法に応用することである

第 A 章

定理証明器HOL

HOL は Cambridge 大学の M.C.J Gordon らによって開発された高階述語論理の定理証明器である．型理論としては Church の単純型理論 ($\lambda\rightarrow$-Church) に最も近い．実用性が高く，純粋な数学理論の証明からハード ウェアの検証に至るまで幅広く適用されている．最新版は HOL4 Kananaskis-3 である．

A.1 メタ言語 moscowML

HOL は moscowML[3] 上に実装されており，論理要素は ML のデータ型として扱われる．例えば，型，項，定理はそれぞれ，`holtype`, `term`, `thm` という型を持つ．推論規則やタクティックも ML の関数として実装されている．

A.2 HOLにおける理論 (theory)

HOL において理論は，型, 定数, 公理, 定理の 4 つの要素から構成され，ML のモジュールにまとめられる．例えば整数の理論 `int` はファイル `intTheory.sml` にまとめられている．理論には親子関係があり，ルートに位置するのが `min` と呼ばれる理論である．すべての理論は `min` を拡張していくことにより構築されている．ここで理論の拡張とは，定義の導入や定理の証明を行うことである．HOL 起動時にユーザに与えられる理論は `bool` という理論であり，これを拡張することにより，ユーザ独自の理論を生成することが可能である．

A.3　証明法

証明はインタプリタと対話的に行われる．証明法には，Forward Proof と Goal Oriented Proof の 2 種類がある．Forward Proof は公理とすでに証明された定理に導出規則を適用して新たな定理を導くという証明法である．Goal Oriented Proof は，証明対象の命題とその前提条件をゴールに設定し，タクティックにより分解していくという証明法である．

A.4　タクティック

タクティックは Goal Oriented Proof におけるゴール分解機能であり，証明の対象となっているゴール G を複数のサブゴール G1,...,Gn に分解する．ML においては tactic という型を持つ関数として実装される．以下に基本的なタクティックを示す．ゴールは仮定と命題の組であり，ここでは [u]v と表記する．

- CONJ_TAC : tactic

  ゴール [u]P/\Q を 2 つのサブゴール [u]P, [u]Q に分解する．

- EQ_TAC : tactic

  ゴール [u]P=Q を 2 つのサブゴール [u]P==>Q, [u]B==>A に分解する．

- INDUCT_TAC : tactic

  自然数に関する命題を数学的帰納法の初期段階と帰納段階に分解する．つまり, ゴール [u]!n.P を 2 つのサブゴール [u]P[0/n], [u,P]P[SUC n'/n] に分解する.

- REWRITE_TAC : thm list -> tactic

  定理のリスト [T1,...,Tn] を引数にとり，T1,...,Tn によりゴールの書き換えを行う．

HOL の最新版には高度なリダクションを行う RW_TAC や PROVE_TAC などのタクティックが導入されている．

A.5　タクティカル

タクティカルは既存のタクティックを組み合わせてより強力なタクティックを構築するための機能である．証明のパターンに応じてタクティックを組み合わせることにより，そのパターンを一度に証明するタクティックを作ることができる．以下，基本的なタクティカルを示す．

- `THEN : tactic -> tactic -> tactic`

  タクティックの逐次適用を実現する．T1,T2をタクティックとすると，`T1 THEN T2`は,「まず，ゴールにT1を適用し，生成されるすべてのサブゴールにT2を適用する」というタクティックとなる．

- `THENL : tactic -> tactic list -> tactic`

  タクティックの選択的逐次適用を実現する．T1,...,Tnをタクティックとすると，`T1 THENL [T1,...,Tn]`は,「まず，ゴールにT1を適用し，生成されるサブゴールG1,...,Gnに対しそれぞれT1,...,Tnを適用する」というタクティックとなる．

- `REPEAT : tactic -> tactic`

  タクティックの反復適用を実現する．Tをタクティックとすると，`REPEAT T`は,「ゴールにTを適用可能な限り繰り返し適用する」というタクティックとなる．

A.6　ライブラリ

HOLでは，特定の理論の証明を支援するためのタクティックや推論規則がライブラリとして提供されている．例えば`intLib`というライブラリには整数に関する定理を証明するためのタクティック`ARITH_TAC`が提供されている．ユーザが独自のライブラリを作成することも可能である．

A.7　論理記号の表記

この論文で使われる HOL の論理記号を以下にまとめる.

HOL における表記	標準的な表記	意味
T	$\top$	真
F	$\bot$	偽
~	$\neg$	否定
/\	$\wedge$	論理積
\/	$\vee$	論理和
==>	$\Rightarrow$	含意
!	$\forall$	全称
?	$\exists$	存在
\	λ	抽象
#	$*$	直積
FST	Fst	直積の第一要素
SND	Snd	直積の第二要素

表 A.1: 論理記号の表記

第 B章

図書館システムの検証

ここでは，4.4 節に取り上げた図書館システムの検証の詳細を述べる．

B.1 クラスモデル

図 4.9 に示される図書館システムのクラスモデルは理論生成器の入力ファイルとしては次のように定義される．

```
class library

attr max       holtype : num | 0         mltype : int | 0
attr days      holtype : num | 0         mltype : int | 0
attr nextcid   holtype : num | 0         mltype : int | 0
attr nextiid   holtype : num | 0         mltype : int | 0
attr itemlist  holtype : item list | []  mltype : item list | []
attr lendlist  holtype : lend list | []  mltype : lend list | []
attr customerlist
          holtype : customer list | []  mltype : customer list | []

class customer

attr cid       holtype : num      | 0   mltype : int      | 0
attr name      holtype : string   | ""  mltype : string   | ""
attr lendlist  holtype : lendlist | []  mltype : lendlist | []
```

```
class item

attr iid    holtype : num    | 0          mltype : int      | 0
attr title  holtype : string | ""         mltype : string   | ""
attr lend   holtype : lend   | lend_null  mltype : lend     | lend_null

class book extends item

attr isbn  holtype : num | 0   mltype : int | 0

class cd extends item

class lend

attr days      holtype : num   | 0          mltype : int   | 0
attr item      holtype : item  | item_null  mltype : item  | item_null
attr customer
    holtype : customer | customer_null  mltype : customer  | customer_null
```

B.2 証明の詳細

貸出手続きが不変表明 Inv1 を維持すること，つまり，

```
!lib cid iid s.
    Inv1 lib s /\ Inv2 lib s /\ Inv3 lib s ==>
        Inv1 lib (SND (library_lend lib cid iid s))
```

の証明について述べる．ここで，Inv1 は図 4.13 において定義される述語であり，library_lendは図 4.11, 図 4.12 において定義される関数である．また，Inv2, Inv3 は Inv1 の証明の前提となる不変表明であり，次のように定義される．

```
Inv2 : library -> store -> bool
Inv2 lib s = library_ex lib s ==>
    let l = library_get_customerlist lib s in
        !x. MEM x l ==> customer_ex x s /\ (COUNT x l = 1)
Inv3 : library -> store -> bool
Inv3 lib s = library_ex lib s ==>
```

```
    let l = library_get_itemlist lib s in
        !x. MEM x l ==> item_ex x s /\ (COUNT x l = 1)
```

Inv2, Inv3はそれぞれ，libraryオブジェクトの保持するcustomerオブジェクトのリスト，itemオブジェクトのリストについて，その要素はストアに存在し，かつ，同じ要素が重複して存在しないことを意味する．COUNT x lはリストlにおける要素xの出現回数である．

Inv1は,「貸出手続き前後で全利用者の貸出数の総和は貸出中の物品の総数に等しい」を意味する不変表明である．この表明が成立する根拠となるのが,「貸出手続きが実行された後，貸出を行った利用者の貸出数は 1 増加し，それに合わせて全利用者の貸出数の総和が 1 増加する」という性質と「貸出手続きが実行された後，貸出の対象となった物品は貸出中となり，それに合わせて貸出中の物品の総数が 1 増加する」という性質である．

これら 2 つの性質を一般化したものが次の 2 つの定理である．

```
|- !(x:'a) f g l.
    ((COUNT x l = 1) /\ (f x = g x + 1) /\
        !y. ~(x = y) ==> (f y = g y)) ==>
            (SUM (MAP f l) = SUM (MAP g l) + 1)

|- !(x:'a) f g l.
    ((COUNT x l = 1) /\ f x /\ ~(g x) /\
        !y. ~(x = y) ==> (f y = g y)) ==>
            (LENGTH (FILTER f l) = LENGTH (FILTER g l) + 1)
```

1 つ目の定理の意味は次の通りである.「いま，あるオブジェクトxがリストlに 1 つだけ存在し（前提条件の 1 つ目），このxに関数fを適用して得られる値が，関数gを適用して得られる値より 1 だけ大きい（前提条件の 2 つ目）とする．また，xとは異なるすべてのオブジェクトに対しては，fを適用して得られる値とgを適用して得られる値は等しいとする（前提条件の 3 つ目）．このとき，lに含まれるすべてのオブジェクトに対してfを適用して得られる値の総和はgを適用して得られる値の総和より 1 だけ大きい.」

2 つ目の定理の意味は次の通りである.「いま，あるオブジェクトxがリストlに 1 つだけ存在し（前提条件の 1 つ目），このxに関数fを適用して得られる値が真

であり（前提条件の 2 つ目），関数 g を適用して得られる値が偽である（前提条件の 3 つ目）とする．また，x とは異なるすべてのオブジェクトに対しては，f を適用して得られる値と g を適用して得られる値は等しいとする（前提条件の 3 つ目）．このとき，l に含まれるすべてのオブジェクトに対して f を適用して真となるオブジェクトの数は，g を適用して真となるオブジェクトの数より 1 だけ大きい.」図 B.1 はこれらの定理を図示したものである．

1 つ目のの定理の x, f, g, l をそれぞれ

```
library_get_customer lib cid s
\x. customer_get_lendnum x (SND (library_lend lib cid iid s))
\x. customer_get_lendnum x s
library_get_customerlist lib s
```

に特化し，2 つ目のの定理の x, f, g, l をそれぞれ

```
library_get_item lib iid s
\x. ~item_is_available x (SND (library_lend lib cid iid s))
\x. ~item_is_available x s
library_get_itemlist lib s
```

に特化すれば，図書館システムの場合に当てはまる．証明は，まずこれら 2 つの定理の前提条件を補題として証明していき，最後にそれらの補題から目的の定理を導出する，という方針で行っている．以下，実際に行った証明のコードを示す．図 B.2 には補題の依存関係を示す．

```
(*** 1 ***)
val MEM_MAP_MEM_HD_FILTER = prove
    (‘‘!x f l. MEM x (MAP f l) ==>
     MEM (HD (FILTER (\y. f y = x) l)) l‘‘,
     Induct_on ‘l‘ THEN RW_TAC list_ss [listTheory.FILTER]);;

(*** 2 ***)
(* 貸出条件が成立するならば，貸出を行う利用者は図書館に登録されている *)
val library_check_lend_MEM_library_get_customer = prove
    (‘‘!lib cid iid s. library_check_lend lib cid iid s ==>
     MEM (library_get_customer lib cid s) (library_get_customerlist lib s)‘‘,
     REWRITE_TAC [library_check_lend] THEN LET_TAC THEN
```

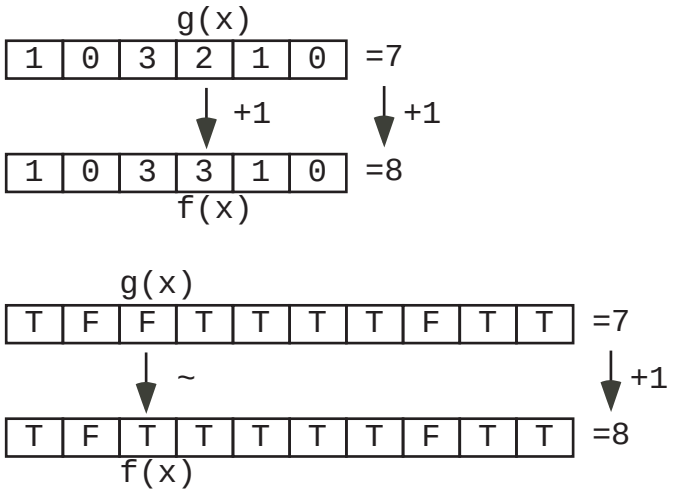


図 B.1: 貸出前後の状態変化

```
    RW_TAC bool_ss [] THEN
    ASSUM_LIST (fn thl => THM_UNDISCH_TAC (el 5 thl)) THEN
    REWRITE_TAC [library_check_cid] THEN LET_TAC THEN
    REWRITE_TAC [library_get_customer] THEN LET_TAC THEN
    SIMP_TAC bool_ss [MEM_MAP_MEM_HD_FILTER]);;

(*** 3 ***)
(* 貸出条件が成立するならば，貸出を行う利用者オブジェクトはストアに存在する *)
val library_check_lend_customer_ex = prove
    (``!lib cid iid s. Inv2 lib s ==> library_ex lib s ==>
     library_check_lend lib cid iid s ==>
          customer_ex (library_get_customer lib cid s) s``,
     EVERY
     [REWRITE_TAC [Inv2], LET_TAC, RW_TAC bool_ss [],
      IMP_RES_TAC library_check_lend_MEM_library_get_customer,
      RES_TAC]);;

(*** 4 ***)
(* 図書館に登録される利用者リストは貸出手続きを行っても変化しない *)
val library_get_customerlist_library_lend = prove
    (``!lib cid iid s.
     library_get_customerlist lib (SND (library_lend lib cid iid s)) =
```

```
     library_get_customerlist lib s‘‘,
     EVERY
     [RW_TAC list_ss [library_lend],
      LET_PAIR_TAC, RW_TAC list_ss [new_lend], LET_PAIR_TAC,
      REWRITE_TAC [library_add_lend,customer_add_lend],
      LET_TAC, SLICE_TAC]);;

(*** 5 ***)
(* 貸出手続き後, 貸出を行った利用者の貸出数は 1 増加する *)
val customer_get_lendnum_library_lend = prove
    (‘‘!lib cid iid s. Inv2 lib s ==> library_ex lib s ==>
     library_check_lend lib cid iid s ==>
      (customer_get_lendnum (library_get_customer lib cid s)
        (SND (library_lend lib cid iid s)) =
           customer_get_lendnum (library_get_customer lib cid s) s + 1)‘‘,
     EVERY
     [RW_TAC bool_ss [library_lend], LET_PAIR_TAC,
      REWRITE_TAC [new_lend], LET_PAIR_TAC,
      REWRITE_TAC [library_add_lend,customer_add_lend,customer_get_lendnum],
      LET_TAC, SLICE_TAC,
      IMP_RES_TAC library_check_lend_customer_ex,
      OBJ_SIMP_TAC, RW_TAC list_ss []]);;

(*** 6 ***)
(* 貸出手続き後, 貸出を行った利用者以外の利用者の貸出数は変化しない *)
val diff_customer_get_lendnum_library_lend = prove
    (‘‘!lib cid iid s cst.
     ~(library_get_customer lib cid s = cst) ==>
      (customer_get_lendnum cst (SND (library_lend lib cid iid s)) =
       customer_get_lendnum cst s)‘‘,
     EVERY
     [RW_TAC list_ss [library_lend], LET_PAIR_TAC,
      REWRITE_TAC [new_lend], LET_PAIR_TAC,
      REWRITE_TAC [library_add_lend,customer_add_lend,customer_get_lendnum],
      LET_TAC, SLICE_TAC, OBJ_SIMP_TAC]);;
```

```
(*** 7 ***)
local
    val lemma1 = prove
    (‘‘!(x:’a) l. (COUNT x l = 0) ==> !y. MEM y l ==> ~(x = y)‘‘,
     RW_TAC list_ss [COUNT,NOT_MEM_COUNT]);;
    val lemma2 = prove
    (‘‘!(x:’a) l.
     ((COUNT x l = 0) /\ !y. ~(x = y) ==> (f y = g y))
            ==> (SUM (MAP f l) = SUM (MAP g l))‘‘,
     Induct_on ‘l‘ THEN RW_TAC list_ss [COUNT]);;
in
    val SUM_MAP_lemma = prove
    (‘‘!(x:’a) f g l.
     ((COUNT x l = 1) /\ (f x = g x + 1) /\
        !y. ~(x = y) ==> (f y = g y))
               ==> (SUM (MAP f l) = SUM (MAP g l) + 1)‘‘,
     Induct_on ‘l‘ THENL
     [RW_TAC list_ss [COUNT],
      EVERY
      [RW_TAC list_ss [COUNT],
       IMP_RES_TAC lemma1, IMP_RES_TAC lemma2]])
end;;

(*** 8 ***)
(* 貸出条件が成立するならば,
   貸出を行う利用者は図書館の利用者リストに重複して含まれていない *)
val library_check_lend_COUNT_library_get_customer = prove
    (‘‘!lib cid iid s. Inv2 lib s ==> library_ex lib s ==>
     library_check_lend lib cid iid s ==>
        (COUNT (library_get_customer lib cid s)
                           (library_get_customerlist lib s) = 1)‘‘,
     EVERY
     [REWRITE_TAC [LET_RULE Inv2], RW_TAC bool_ss [],
      IMP_RES_TAC library_check_lend_MEM_library_get_customer,
      RES_TAC]);;
```

```
(*** 9 ***)
(* 貸出手続きが成功したとき，全利用者の貸出総数は 1 増加する *)
local
    local
        val t_x = ‘‘library_get_customer lib cid s‘‘;
        val t_f =
           ‘‘\x. customer_get_lendnum x
               (SND (library_lend lib cid iid s))‘‘;
        val t_g = ‘‘\x. customer_get_lendnum x s‘‘;
        val t_l = ‘‘library_get_customerlist lib s‘‘;
    in
        val lemma =
            BETA_RULE
            (SPECL [t_x,t_f,t_g,t_l]
            (INST_TYPE [{redex=‘‘:’a‘‘,residue=‘‘:customer‘‘}] SUM_MAP_lemma))
    end
in
    val library_get_customer_lendsum_library_lend_1 = prove
    (‘‘!lib cid iid s.
     Inv2 lib s ==> library_ex lib s ==>
       library_check_lend lib cid iid s ==>
        (library_get_customer_lendsum lib (SND (library_lend lib cid iid s)) =
         library_get_customer_lendsum lib s + 1)‘‘,
     EVERY
     [REWRITE_TAC [library_get_customer_lendsum], LET_TAC,
      REWRITE_TAC [library_get_customerlist_library_lend],
      RW_TAC bool_ss [],
      IMP_RES_TAC library_check_lend_COUNT_library_get_customer,
      IMP_RES_TAC customer_get_lendnum_library_lend,
      ASSUME_TAC
      (SPECL [‘‘lib:library‘‘,‘‘cid:num‘‘,‘‘iid:num‘‘,‘‘s:store‘‘]
       diff_customer_get_lendnum_library_lend),
       RW_TAC bool_ss [lemma]])
end;;

(*** 10 ***)
```

```
(* 貸出手続きが失敗したとき，全利用者の貸出総数は変化しない *)
val library_get_customer_lendsum_library_lend_2 = prove
    (‘‘!lib cid iid s. ~(library_check_lend lib cid iid s) ==>
     (library_get_customer_lendsum lib (SND (library_lend lib cid iid s)) =
      library_get_customer_lendsum lib s)‘‘,
     RW_TAC list_ss [library_lend]);;

(*** 11 ***)
(* 貸出条件が成立するならば，貸出対象の物品は図書館に登録されている *)
val library_check_lend_MEM_library_get_item = prove
    (‘‘!lib cid iid s. library_check_lend lib cid iid s ==>
     MEM (library_get_item lib iid s) (library_get_itemlist lib s)‘‘,
     EVERY
     [REWRITE_TAC [library_check_lend], LET_TAC, RW_TAC bool_ss [],
      ASSUM_LIST (fn thl => THM_UNDISCH_TAC (el 4 thl)),
      REWRITE_TAC [library_check_iid], LET_TAC,
      REWRITE_TAC [library_get_item], LET_TAC,
      SIMP_TAC bool_ss [MEM_MAP_MEM_HD_FILTER]]);;

(*** 12 ***)
(* 貸出条件が成立するならば，貸出対象の物品オブジェクトはストアに存在する *)
val library_check_lend_item_ex = prove
    (‘‘!lib cid iid s. Inv3 lib s ==> library_ex lib s ==>
     library_check_lend lib cid iid s ==>
         item_ex (library_get_item lib iid s) s‘‘,
     EVERY
     [REWRITE_TAC [LET_RULE Inv3], RW_TAC bool_ss [],
      IMP_RES_TAC library_check_lend_MEM_library_get_item,
      RES_TAC]);;

(*** 13 ***)
(* 図書館に登録される物品リストは貸出手続きを行っても変化しない *)
val library_get_itemlist_library_lend = prove
    (‘‘!lib cid iid s.
     library_get_itemlist lib (SND (library_lend lib cid iid s)) =
     library_get_itemlist lib s‘‘,
```

```
     EVERY
     [RW_TAC list_ss [library_lend], LET_PAIR_TAC,
      RW_TAC list_ss [new_lend], LET_PAIR_TAC,
      REWRITE_TAC [library_add_lend,customer_add_lend],
      LET_TAC, SLICE_TAC]);;

(*** 14 ***)
(* 貸出手続き後, 貸出対象の物品は貸出中となる *)
val item_is_available_library_lend = prove
     (``!lib cid iid s. Inv3 lib s ==> library_ex lib s ==>
      library_check_lend lib cid iid s ==>
          ~(item_is_available (library_get_item lib iid s)
               (SND (library_lend lib cid iid s)))``,
      EVERY
      [RW_TAC list_ss [library_lend], LET_PAIR_TAC,
       REWRITE_TAC [new_lend], LET_PAIR_TAC,
       REWRITE_TAC [library_add_lend,customer_add_lend], LET_TAC,
       REWRITE_TAC [item_is_available], SLICE_TAC,
       IMP_RES_TAC library_check_lend_item_ex,
       OBJ_SIMP_TAC]);;

(*** 15 ***)
(* 貸出手続き後, 貸出対象の物品以外の物品の貸出可能性は変化しない *)
val diff_item_is_available_library_lend = prove
     (``!lib cid iid s itm. ~(library_get_item lib iid s = itm) ==>
      (item_is_available itm (SND (library_lend lib cid iid s)) =
       item_is_available itm s)``,
      EVERY
      [RW_TAC list_ss [library_lend], LET_PAIR_TAC,
       REWRITE_TAC [new_lend], LET_PAIR_TAC,
       REWRITE_TAC [library_add_lend,customer_add_lend],
       LET_TAC, REWRITE_TAC [item_is_available],
       SLICE_TAC, OBJ_SIMP_TAC]);;

(*** 16 ***)
local
```

```
    val lemma1 = prove
        (‘‘!(x:’a) f g l. (COUNT x l = 0) /\ (!y. ~(x = y) ==> (f y = g y))
             ==> (LENGTH (FILTER f l) = LENGTH (FILTER g l))‘‘,
         GEN_TAC THEN GEN_TAC THEN GEN_TAC THEN
         Induct_on ‘l‘ THENL
         [RW_TAC list_ss [listTheory.FILTER],
          GEN_TAC THEN Cases_on ‘x=h‘ THEN
          RW_TAC list_ss [COUNT,listTheory.FILTER]])
in
    val LENGTH_FILTER_lemma = prove
        (‘‘!(x:’a) f g l.
            ((COUNT x l = 1) /\ f x /\ ~(g x) /\
               !y. ~(x = y) ==> (f y = g y)) ==>
                  (LENGTH (FILTER f l) = LENGTH (FILTER g l) + 1)‘‘,
         GEN_TAC THEN GEN_TAC THEN GEN_TAC THEN
         Induct_on ‘l‘ THENL
         [RW_TAC list_ss [COUNT],
          GEN_TAC THEN Cases_on ‘x=h‘ THENL
          [RW_TAC list_ss [COUNT,listTheory.FILTER] THEN
          IMP_RES_TAC lemma1 THEN ASM_SIMP_TAC arith_ss [],
          RW_TAC list_ss [COUNT,listTheory.FILTER]]])
end;;

(*** 17 ***)
(* 貸出条件が成立するならば,
   貸出対象の物品は図書館の物品リストに重複して含まれていない *)
val library_check_lend_COUNT_library_get_item = prove
    (‘‘!lib cid iid s. Inv3 lib s ==> library_ex lib s ==>
     library_check_lend lib cid iid s ==>
       (COUNT (library_get_item lib iid s) (library_get_itemlist lib s) = 1)‘‘,
     EVERY
     [REWRITE_TAC [LET_RULE Inv3], RW_TAC bool_ss [],
      IMP_RES_TAC library_check_lend_MEM_library_get_item,
      RES_TAC]);;

(*** 18 ***)
```

```
(* 貸出条件が成立するならば，貸出対象の物品は貸出可能状態にある *)

val library_check_lend_item_is_available = prove
     (‘‘!lib cid iid s. library_check_lend lib cid iid s ==>
      item_is_available (library_get_item lib iid s) s‘‘,
      EVERY
      [REWRITE_TAC [library_check_lend], LET_TAC, RW_TAC bool_ss []]);;

(*** 19 ***)
(* 貸出手続きが成功したとき，貸出中の物品の総数は 1 増加する *)
local
    local
        val t_x = ‘‘library_get_item lib iid s‘‘;;
        val t_f =
            ‘‘\x. ~item_is_available x (SND (library_lend lib cid iid s))‘‘;;
        val t_g = ‘‘\x. ~item_is_available x s‘‘;;
        val t_l = ‘‘library_get_itemlist lib s‘‘;;
    in
        val lemma =
            REWRITE_RULE []
            (BETA_RULE
            (SPECL [t_x,t_f,t_g,t_l]
            (INST_TYPE [{redex=‘‘:’a‘‘,residue=‘‘:item‘‘}]
             LENGTH_FILTER_lemma)))
    end
in
    val library_get_item_lendsum_library_lend_1 = prove
    (‘‘!lib cid iid s.
       Inv3 lib s ==> library_ex lib s ==>
       library_check_lend lib cid iid s ==>
         (library_get_item_lendsum lib (SND (library_lend lib cid iid s)) =
          library_get_item_lendsum lib s + 1)‘‘,
     EVERY
     [REWRITE_TAC [library_get_item_lendsum], LET_TAC,
      REWRITE_TAC [library_get_itemlist_library_lend],
      RW_TAC bool_ss [],
```

```
      IMP_RES_TAC library_check_lend_COUNT_library_get_item,
      IMP_RES_TAC library_check_lend_item_is_available,
      IMP_RES_TAC item_is_available_library_lend,
      ASSUME_TAC
      (SPECL [``lib:library``,``cid:num``,``iid:num``,``s:store``]
       diff_item_is_available_library_lend),
      ASM_SIMP_TAC bool_ss [lemma]])
end;;

(*** 20 ***)
(* 貸出手続きが失敗したとき，貸出中の物品の総数は変化しない *)
val library_get_item_lendsum_library_lend_2 = prove
    (``!lib cid iid s. ~(library_check_lend lib cid iid s) ==>
     (library_get_item_lendsum lib (SND (library_lend lib cid iid s)) =
      library_get_item_lendsum lib s)``,
     RW_TAC list_ss [library_lend]);;

(*** 21 ***)
(* 図書館オブジェクトがストアに存在することは貸出手続きの実行に無関係である *)
val library_ex_library_lend = prove
    (``!lib cid iid s.
     library_ex lib (SND (library_lend lib cid iid s)) = library_ex lib s``,
     EVERY
     [RW_TAC list_ss [library_lend], LET_PAIR_TAC,
      REWRITE_TAC [new_lend], LET_PAIR_TAC,
      REWRITE_TAC [library_add_lend,customer_add_lend],
      LET_TAC, SLICE_TAC]);;

(*** 22 ***)
(* 貸出手続きが不変表明を維持する *)
val Inv1_library_lend = prove
    (``!lib cid iid s.
       Inv1 lib s /\ Inv2 lib s /\ Inv3 lib s ==>
         Inv1 lib (SND (library_lend lib cid iid s))``,
     EVERY
     [REWRITE_TAC [Inv1], REWRITE_TAC [library_ex_library_lend],
```

```
RW_TAC bool_ss [],
Cases_on ‘library_check_lend lib cid iid s‘ THENL
[ASM_SIMP_TAC arith_ss
 [library_get_customer_lendsum_library_lend_1,
  library_get_item_lendsum_library_lend_1],
 ASM_SIMP_TAC arith_ss
 [library_get_customer_lendsum_library_lend_2,
  library_get_item_lendsum_library_lend_2]]);;
```

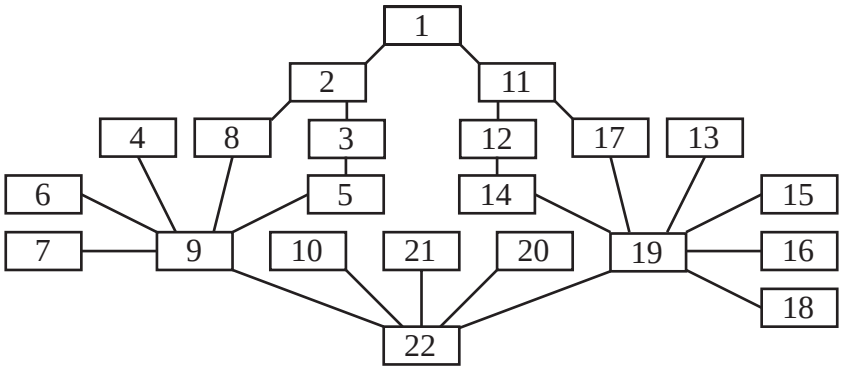


図 B.2: 補題の依存グラフ

謝辞

本研究を行うにあたり，終始御指導賜った片山卓也教授に深く感謝致します．また，日頃から有益な助言を頂いた青木利晃助手に感謝致します．研究室の諸兄には，ゼミ活動など広い範囲にわたり議論の相手をして頂いたことに感謝致します．

参考文献

[1] OMG. Unified Modeling Language.
URL: http://www.omg.org/.

[2] The HOL system.
URL: http://hol.sourceforge.net/.

[3] Moscow ML.
URL: http://www.dina.dk/ sestoft/mosml.html.

[4] J. Warmer and A. Kleppe. The object constraint language: precise modeling with UML. Addison-Wesley, 1999.

[5] M. Abadi and L.Cardeli. A theory of primitive objects. Untyped and first-order systems. Information and Computation, 125(2):78-102, 1996.

[6] Tobias Nipkow, David von Oheimb and Cornelia Pusch. μJava: Embedding a Programming Language in a Theorem Prover. In Foundations of Secure Computation. IOS Press, 2000.

[7] Bart Jacobs et al. LOOP project, http://www.cs.kun.nl/ bart/LOOP/

[8] J. van den Berg, M. Huisman, B. Jacobs, and E. Poll. A type-theoretic memory model for verification of sequential Java programs. Techn. Rep. CSI-R9924, Comput. Sci. Inst., Univ. of Nijmegen, 1999.

[9] David von Oheimb. Hoare Logic for Java in Isabelle/HOL. Concurrency and Computation: Practice and Experience, vol.13 pp.1173-1214, 2001.

[10] Claude Marché and Christine Paulin-Mohring. Reasoning on Java programs with aliasing and frame conditions. In 18th International Conference on Theorem Proving in Higher Order Logics (TPHOLs 2005), LNCS, August 2005.

[11] W. Naraschewski and M. Wenzel. Object-Oriented Verification based on Record Subtyping in Higher-Order Logic. Tecnische Universitat Munchen, 1998.

[12] A. Poetzsch-Heffer and P. Muller. Logical Foundations for Typed Object-Oriented Languages. Programming Concepts and Methods (PROCOMET), 1998.

[13] E.M.Clarke and W.Heinle. Modular Translation of Satatecharts to SMV. URL: http://iamwww.unibe.ch/ ftiwww/Personen /heinle/papers.html.

[14] T.Schafer, A.Knapp, and S.Merz. Model Checking UML State Machines and Collaborations. On the Workshop on Software Model Checking, July 2001. Electronic Notes in Theoretical Computer Science vol.55, n.3.

[15] Using B formal specifications for analysis and verfication of UML/OCL models. Marcano, R. and N. Levy. Workshop on consistency problems in UML-based software development. 5th International Conference on the Unified Modeling Language. Dresden, Germany, October 2002.

[16] K. Lano, D. Clark and K. Androutsopoulos. UML to B: Formal Verification of Object-Oriented Models. Integrated Formal Methods: 4th International Conference, IFM 2004, Cnaterbury, UK, April 4-7, 2004.

[17] Christian Prehofer. Feature-oriented programming: A fresh look at objects. In Proceedings of ECOOP'97. Springer-LNCS, 1997.

[18] D. Batory, R. Cardone, and Y. Smaragdakis. Object-Oriented Frameworks and Product-Lines. 1st Software Product-Line Conference. Denver Colorado, August 1999.

[19] Harry Li, Shriram Krishnamurthi, and Kathi Fisler. Verifying cross-cutting features as open systems. In international conference on Foundation of Software Engineering, 2002.

[20] Kathi Fisler and Shriram Krishnamurthi. Modular verification of collaboration-based software designs. In Symposium on the Foundation of Software Engineering, 2001.

[21] Y. Smaragdakis and D. Batory. Implementing layered designs with mixin layers. Proceedings of the European Conference on Object-Oriented Programming (ECOOP), 1998.

[22] Brian Roberts. Modular Detection of Feature Interactions Through Theorem Proving: A Case Study. A Thesis submitted to the faculty of the Worcester Polytechnic Institute, 2003.

[23] D. B. Aredo. Semantics of UML sequence diagrams in PVS. In Proc. of UML2000 Workshop on Dynamic Behavior in UML Models: Semantic Questions, October 2000, York, UK.

[24] R. Heckel and S. Sauer. Strengthenig the Semantics of UML Collaboration Diagrams. In Proc. of UML2000 Workshop on Dynamic Behavior in UML Models: Semantic Questions, October 2000, York, UK.

[25] 青木利晃, 立石 孝彰, 片山卓也. 定理証明技術のオブジェクト指向分析への適用. 日本ソフトウェア科学会学会誌 コンピュータソフトウェア, Vol.18, No.4, pp.18-47, 2001.

[26] 立石孝彰，青木利晃，片山卓也. 振舞い近似手法を用いたステートチャートに対する不変性の検証. 情報処理学会論文誌, Vol. 44, No. 6, pp. 1448-1460, 2003.

本研究に関する発表論文

[1] Kenro Yatake, Toshiaki Aoki and Takuya Katayama: Implementing application-specific Object-Oriented theories in HOL, International Conference on Theoretical Aspects of Computing, ICTAC 2005.

[2] Kenro Yatake, Toshiaki Aoki and Takuya Katayama: Collaboration-based Verfication of Object-Oriented models in HOL, Proceedings of the 2nd International Workshop on Verification and Validation of Enterprise Information Systems, VVEIS 2004, pp.78-80, 2004.

[3] 矢竹健朗, 青木利晃, 片山卓也：コラボレーションに基づくオブジェクト指向モデルの検証, コンピュータソフトウェア, Vol.22, No.1(2005), pp.58-76.

[4] 矢竹健朗，青木利晃，片山卓也：定理証明システム HOL におけるオブジェクト指向理論の構築，情報処理学会 ソフトウェア工学研究会 研究報告 2002-SE-138, 2002.